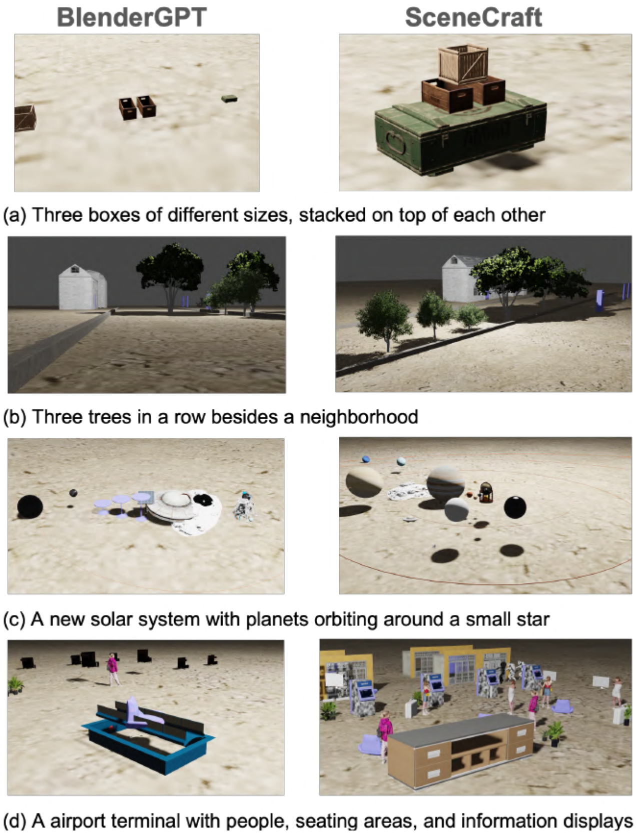
SceneCraft: An LLM Agent for Synthesizing 3D Scenes as Blender Code

Ziniu Hu [1 2] Ahmet Iscen [2] Aashi Jain [2] Thomas Kipf [2] Yisong Yue [1] David A Ross [2] Cordelia Schmid [2] Alireza Fathi [2]

Abstract

This paper introduces SceneCraft, a Large Language Model (LLM) Agent converting text descriptions into Blender-executable Python scripts which render complex scenes with up to a hundred 3D assets. This process requires complex spatial planning and arrangement. We tackle these challenges through a combination of advanced abstraction, strategic planning, and library learning. SceneCraft first models a scene graph as a blueprint, detailing the spatial relationships among assets in the scene. SceneCraft then writes Python scripts based on this graph, translating relationships into numerical constraints for asset layout. Next, SceneCraft leverages the perceptual strengths of vision-language foundation models like GPT-V to analyze rendered images and iteratively refine the scene. On top of this process, SceneCraft features a library learning mechanism that compiles common script functions into a reusable library, facilitating continuous self-improvement without expensive LLM parameter tuning. Our evaluation demonstrates that SceneCraft surpasses existing LLM-based agents in rendering complex scenes, as shown by its adherence to constraints and favorable human assessments. We also showcase the broader application potential of SceneCraft by reconstructing detailed 3D scenes from the Sintel movie and guiding a video generative model with generated scenes as intermediary control signal.



(a) Three boxes of different sizes, stacked on top of each other

(b) Three trees in a row besides a neighborhood

(c) A new solar system with planets orbiting around a small star

(d) A airport terminal with people, seating areas, and information displays

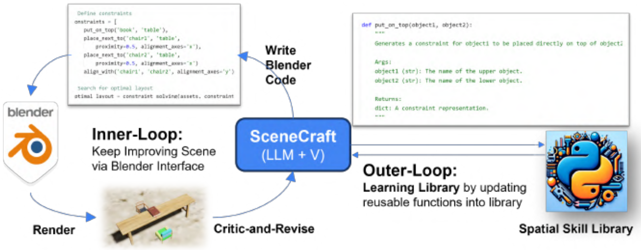
Figure 1. Examples comparing SceneCraft's output against a BlenderGPT baseline for different queries.

1. Introduction

Transforming natural language descriptions into 3D scenes is a key technology for industries like architectural design, game development, virtual reality, and cinematic production. Recent 3D generative models like DreamFusion (Poole et al., 2022) and Magic3D (Lin et al., 2023) have made great progress in transforming text to a 3D neural representation of an object. However, these works fall short of composing entire scenes with multiple assets due to dataset scale limitations and domain specificity. In this work, we are inspired by how human artists typically adopt a holistic process for designing 3D scenes, where they take an iterative, step-by-step approach that includes storyboarding, 3D modeling, texturing, rigging, layout, animation, and rendering, using professional software such as Blender (ble). This iterative process grants the artists in studios a nuanced control over each asset's placement and movement — a level of control not yet achieved by existing models.

[1] California Institute of Technology [2] Google DeepMind. Correspondence to: Ziniu Hu <acgbull@gmail.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).



```
 Define constraints
onstraints = [
   put_on_top('book', 'table'),
   place_next_to('chair1', 'table',
       proximity=0.5, alignment_axes='x'),
   place_next_to('chair2', 'table',
       proximity=0.5, alignment_axes='x')
   align_with('chair1', 'chair2', alignment_axes='y')

 Search for optimal layout
otimal layout - constraint solving(assets, constraint
```

```
def put_on_top(object1, object2):
    """
    Generates a constraint for object1 to be placed directly on top of object2

    Args:
    object1 (str): The name of the upper object.
    object2 (str): The name of the lower object.

    Returns:
    dict: A constraint representation.
    """
```

Figure 2. SceneCraft is composed of a *dual-loop* self-improving pipeline: in the inner-loop, per each scene, an LLM autonomously writes a script to interact with Blender, receives rendered image, and keeps improving the script until getting good scenes; in the outer-loop, SceneCraft summarizes common functions over a batch of written scripts to maintain a reusable design skill library.

Our paper introduces SceneCraft, an LLM-powered agent that is designed to streamline this text-to-3D scene conversion process, closely emulating the workflow of studio artists. SceneCraft transforms textual descriptions into executable Blender code, rendering 3D scenes that are visually cohesive and contextually accurate. This task goes beyond mere data processing, demanding a nuanced understanding of spatial and semantic relationships, which remains a challenge even for today's LLMs. While earlier systems like WordsEye (Coyne & Sproat, 2001) and SceneSere (Chang et al., 2017) have made progress towards using predefined templates and rules to extract spatial constraints from linguistic queries, they depend on extensive human input, especially in new domains with unique relationship patterns.

SceneCraft leverages LLMs to autonomously generate Python code, translating spatial relations within scenes into precise numerical constraints. To achieve this, the core of SceneCraft is a dual-loop optimization pipeline, illustrated in Figure 2. The inner-loop focuses on per-scene layout optimization. Here, an LLM-based planner constructs a scene graph outlining the spatial constraints for asset arrangement. SceneCraft then writes Python code to transform these relations into numerical constraints. These constraints are fed to a specialized solver that determines the layout parameters of each asset, including location, orientation and sizes. After rendering these scripts into images via Blender, we utilize a multimodal LLM (GPT-V (OpenAI, 2023)) to assess the alignment between the generated image and the textual description. If a misalignment is detected, the LLM identifies the problematic semantic relations and corresponding constraints, subsequently refining the scripts. This iterative process of refinement and feedback is crucial for enhancing the scene's fidelity, ensuring each rendition progressively aligns more closely with the original vision, which also matches more the human artists' designing process.

Following the inner-loop refinement of scene scripts, SceneCraft starts its outer loop to dynamically expand its 'spatial skill' library. Within this procedure, it reviews the incremental changes made to the constraint scripts across inner-loop iterations for each scene, identifying and integrating common code patterns, thereby streamlining the acquisition of new non-parametric skills for self-improvement. For instance, if the text query describes a lamp placed on a desk, but the initial rendering shows desk lamps floating mid-air, SceneCraft may learn to introduce a new "grounded" constraint between lamp and desk surfaces. By continuously updating its library through such outer-loop learning over batches, SceneCraft acquires an expanding repertoire of spatial skills over time. SceneCraft is therefore able to handle increasingly complex scenes and descriptions without external human expertise or LLM parameter tuning.

To evaluate SceneCraft, we conduct comprehensive experiments on both synthetic and real-world datasets. First, we create our own curated datasets with ground-truth spatial constraints to quantify SceneCraft's fidelity in translating text to constraint scripts. Second, we apply SceneCraft to the Sintel movie dataset by finetuning a video generative model on the first half of the movie conditioned on ground-truth scene images. For the second half, we generate scenes using SceneCraft and other baselines as input to the video model. Across datasets, results demonstrate SceneCraft's superior sample efficiency and accuracy in rendering intricate 3D scenes from textual descriptions, enabled by its dual-loop optimization. Quantitatively, SceneCraft achieves over 45.1% and 40.9% improvement on generated scenes' CLIP score, compared with another popular LLM agent baseline BlenderGPT, over both unseen synthetic queries and real-world movies like Sintel. SceneCraft also achieves significantly better constraint passing score (88.9 against 5.6). Qualitatively, the scenes and videos generated using SceneCraft more accurately encapsulate the narrative and artistic nuances described in the text. It receives much higher human preference ratings on different perspective, and also benefit a video generative model with very light fine-tuning. Together, our comprehensive evaluation validates SceneCraft as an adaptable and efficient framework for translating imaginative text to 3D reality while continuously improving itself.

This paper's contributions are:

- An LLM Agent that transforms an input text query into a 3D scene by generating a Blender script. The script is iteratively improved by a multimodal LLM that identifies unsatisfied constraints and fixes them in a feedback loop.
- A spatial skill library learned given a set of synthetic input queries without requiring human involvement and LLM fine-tuning, resulting in improved scene generation results.
- Experimental results show that comparing with BlenderGPT, another LLM-based agent baseline,

SceneCraft achieves 45.1% and 40.9% improvement on generated scenes' CLIP score, over both unseen synthetic queries and real-world movies like Sintel.

2. Approach

Our goal is to transform a text query q into a 3D scene s that is not only spatially coherent but also contextually rich and aesthetically pleasing. This requires (a) identifying the correct spatial and contextual relationships between assets, and (b) predicting a high fidelity and nice looking arrangement that aligns with these relationships.

SceneCraft performs this task by building on top of a state-of-the-art multimodal LLM (i.e., GPT-4V (OpenAI, 2023)) and a professional rendering software (Blender). We now describe the key components of our method.

2.1. Asset Retrieval and Scene Decomposition

A scene consists of a set of assets, where each asset a_i is a 3D model. Given the input text query q, the agent makes an LLM call to generate a list of asset names and description that shall be put in the scene. Based on them, a set of 3D assets $\mathcal{A}$ are retrieved from a large repository of 3D objects utilizing a CLIP-based retriever. The retrieval process first finds the top-10 assets based on the text description of each asset. Then each retrieved asset is rendered as an image and the one with the highest text-to-image score is selected.

Some scenes might contain up to a hundred assets, making the layout planning very difficult. Therefore, SceneCraft agent decomposes the scene into a set of sub-scenes, each representing a part of the entire scene. Breaking the problem into small pieces is a widely adopted strategy in natural language question answering (Perez et al., 2020) and general reasoning (Zhou et al., 2023a).

The agent calls a LLM-empowered decomposer that breaks the input query into a sequence of sub-scenes $\hat{s}_k$, each containing a title, a list of asset names $\mathcal{A}_k$ and a sub-scene description q_k. The scene descriptions are used to guide the scene optimization in the later stages.

$$(q_1, \mathcal{A}_1), \ldots, (q_K, \mathcal{A}_K) \leftarrow \texttt{LLM-decomposer}(q). \quad (1)$$

As an example shown in Figure 3, given a query "a girl hunter walking in a slum village with fantasy creatures", SceneCraft decomposes it into three different steps, among which the first step includes the following information:

```
"description":  "In this step, we will set the
     foundation of the scene. The slum street has two
     lines of dilapidated house, each with dim street
     lamps in front of the door. Trash can be found all
     over the street, and one market stall is open at
     the entrance of the street." // This is q_1
"asset_list":  ["Slum house", "Market Stall", "Street
     Lamp", "Trash"] // This is A_1
```

2.2. Scene Graph Construction

In order to put the 3D assets together to create a scene s, each asset a_i requires its corresponding layout matrix $\mathcal{L}(a_i)$, encapsulating the position, scale and orientation of a_i in the scene's coordinate frame. For instance, in a scene described as "a round table with a vase on it, placed near a window", the assets a_i, a_j and a_k could represent the 'table', 'vase' and the 'window', respectively. Their layout matrices $\mathcal{L}(a_i)$, $\mathcal{L}(a_j)$ and $\mathcal{L}(a_k)$ shall position the vase, table and window in the scene such that the vase is standing on the table and the table is located near the window.

The key challenge is to correctly put each asset in the right location and orientation by predicting the layout matrix $\mathcal{L}(a_i)$ for each asset. The naive approach is to directly predict all the layout matrices directly given the scene description. However, this is a highly complex task even for the most advanced LLMs, due to the vast combinatorial space of the possible layouts and deep understanding the intricate spatial relations between assets. This is why in SceneCraft we use a relational scene graph as an intermediate layer of abstraction.

To model the spatial relations between assets, SceneCraft utilizes a set of spatial and contextual relations, such as proximity, alignment, parallelism, etc. Each relation r applies to a specific set of assets within the scene. Full list of relations that we consider can be found in Sec C in Appendix.

Using these relations, the scene s is abstracted into a relational bipartite graph $G(s) = (\mathcal{A}, \mathcal{R}, \mathcal{E})$, which contains two types of nodes: $\mathcal{A}$ represents the set of assets and $\mathcal{R}$ represents the set of relations as nodes. $\mathcal{E}$ represents the edges connecting a relation node to a subset of assets $\mathcal{E}(r)$ in the scene that satisfies this relation.[1]

Based on this definition, SceneCraft then uses a `LLM-Planner` to construct a scene graph connecting assets to corresponding spatial relation nodes.

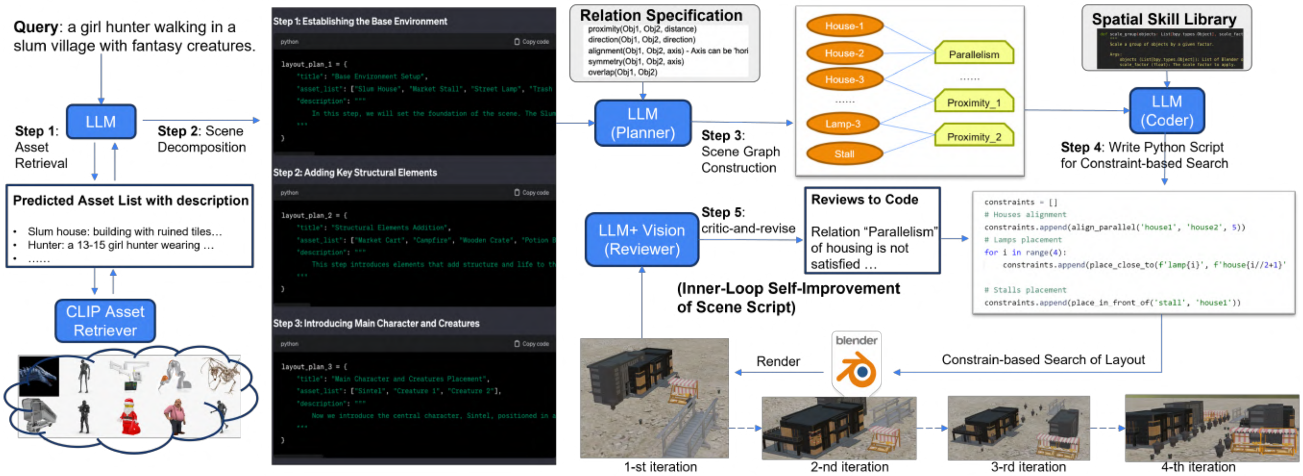
$$\mathcal{G}(s) = (\mathcal{A}, \mathcal{R}, \mathcal{E}) \leftarrow \texttt{LLM-Planner}(q_k, \mathcal{A}) \quad (2)$$

For example, when we create the outline of slum village, `LLM-Planner` predicts the following edges:

- ⟨Alignment, list of houses⟩: all housess are aligned side by side to form a side-street;
- ⟨Parallelism, two list of houses⟩: Duplicate one side of street to form a pathway or road;
- ⟨Proximity, each lamp, each house⟩: lamps are located in front of each house.

The relations between the assets provide soft spatial constraints for the layout matrices $\mathcal{L}$ of the assets. Thus, this

[1] For each relation type, we can have multiple relation nodes linking to different subsets of assets, e.g., *Align-1* relation node links table and vase, and *Align-2* links vase and window.


Query: a girl hunter walking in a slum village with fantasy creatures.
LLM
Step 1: Asset Retrieval
Step 2: Scene Decomposition
Predicted Asset List with description
- Slum house: building with ruined tiles…
- Hunter: a 13-15 girl hunter wearing …
- ……
CLIP Asset Retriever
Relation Specification
proximity(Obj1, Obj2, distance)
direction(Obj1, Obj2, direction)
alignment(Obj1, Obj2, axis) - Axis can be 'hori
symmetry(Obj1, Obj2, axis)
overlap(Obj1, Obj2)
LLM (Planner)
Step 3: Scene Graph Construction
House-1
House-2
House-3
……
Lamp-3
Stall
Parallelism
……
Proximity_1
Proximity_2
Spatial Skill Library
LLM (Coder)
Step 4: Write Python Script for Constraint-based Search
LLM+ Vision (Reviewer)
Step 5: critic-and-revise
Reviews to Code
Relation "Parallelism" of housing is not satisfied …
(Inner-Loop Self-Improvement of Scene Script)
Render
blender
Constrain-based Search of Layout
1-st iteration
2-nd iteration
3-rd iteration
4-th iteration

Step 1: Establishing the Base Environment

```python
layout_plan_1 = {
    "title": "Base Environment Setup",
    "asset_list": ["Slum House", "Market Stall", "Street Lamp", "Trash
    "description": """
        In this step, we will set the foundation of the scene. The Slum
    """
}
```

Step 2: Adding Key Structural Elements

```python
layout_plan_2 = {
    "title": "Structural Elements Addition",
    "asset_list": ["Market Cart", "Campfire", "Wooden Crate", "Potion B
    "description": """
        This step introduces elements that add structure and life to th
    """
}
```

Step 3: Introducing Main Character and Creatures

```python
layout_plan_3 = {
    "title": "Main Character and Creatures Placement",
    "asset_list": ["Sintel", "Creature 1", "Creature 2"],
    "description": """
        Now we introduce the central character, Sintel, positioned in a
    """
}
```

```python
def scale_group(objects: List[bpy.types.Object], scale_fact
    """
    Scale a group of objects by a given factor.

    Args:
        objects (List[bpy.types.Object]): List of Blender o
        scale_factor (float): The scale factor to apply.
```

```python
constraints = []
# Houses alignment
constraints.append(align_parallel('house1', 'house2', 5))
# Lamps placement
for i in range(4):
    constraints.append(place_close_to(f'lamp{i}', f'house{i//2+1}'

# Stalls placement
constraints.append(place_in_front_of('stall', 'house1'))
```

Figure 3. **The workflow of SceneCraft's inner-loop improvement of each scene**. 1) given query, a LLM writes a list of assets descriptions, then use CLIP retriever to fetch assets; 2) then LLM decomposes the full query into a sequence of sub-scene, each associated with a subset of assets and a text description; 3) a LLM-Planner generate a relational graph linking assets to spatial relationship; 4) Based on the graph, LLM-Coder writes python codes to get a list of numerical constraints, which can be executed to search optimal layout, and render into image using Blender; 5) LLM-Reviewer with vision perception capability criticize the rendered image, and update the script accordingly. This critic-and-revise procedure can be done multiple times to iteratively improve the script and scene.

intermediate graph serves as a high-level plan for subsequent code generation and self-improvement, which significantly reduces the complexity of arranging the assets in the scene.

2.3. Scene Layout Optimization in a Feedback Loop

After we obtain the spatial constraints between the assets, we use a set of scoring functions (one per relation) to optimize the scene layout. In Sec. 2.4 we describe how we learn the library of scoring functions automatically. The scoring function $F_r(\cdot)$ for relation r captures whether the constraint is satisfied via $F_r\big(\{\mathcal{L}(a_i) \mid a_i \in \mathcal{E}(r)\}, \mathtt{arg}_r\big)$. F_r takes as input a list of layout matrices $\mathcal{L}(a_i)$ connected by r, as well as the function arguments $\mathtt{arg}_r$ such as the distance, directions, etc. F_r returns a real number between 0 and 1 describing how much this relational constraint is satisfied.

An `LLM-Coder` then reuses these existing functions stored in skill library to synthesize an overall Blender code script `code`, including loading the assets, doing grouping and generating all the numerical constraints, etc. The `LLM-Coder` will also predict all the arguments $\mathtt{arg}_r$ for each function F_r, such as the exact distance for Proximity relation, etc.

$$\mathtt{code}, \mathtt{arg} \leftarrow \mathtt{LLM\text{-}Coder}(\mathcal{G}(s), q_k) \tag{3}$$

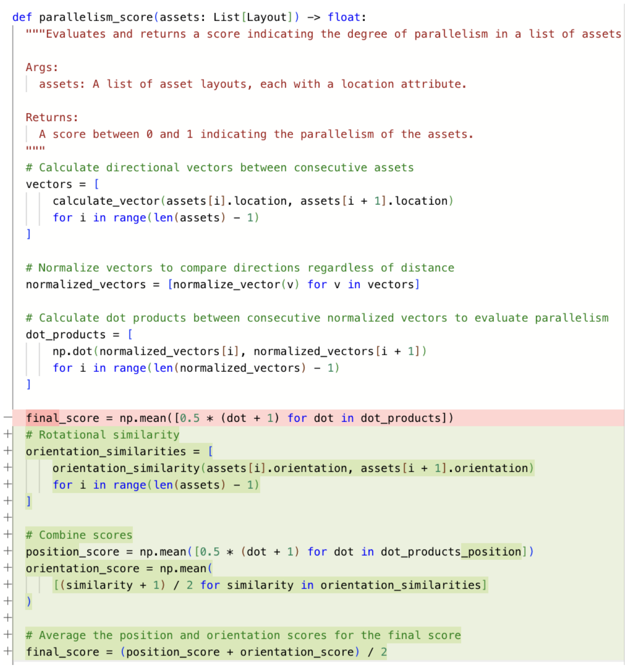
For each scene s abstracted by a relational scene graph $G(s)$ finding an optimal layout could be formalized as the following optimization problem:

$$\hat{\mathcal{L}} \leftarrow \underset{\mathcal{L}}{\mathrm{argmax}} \sum_{r \in \mathcal{R}} F_r\Big(\big\{\mathcal{L}(a_i) \mid a_i \in \mathcal{E}(r)\big\}, \mathtt{arg}_r\Big) \tag{4}$$

This enables SceneCraft to simultaneously balance multiple constraints, ensuring a comprehensive and contextually accurate scene layout planning. After getting the optimal layout $\hat{\mathcal{L}}$, we can render the scene with the Blender code script `code` to get image output. Examples of generated scripts and rendered images can be found at Sec. B in Appendix.

Self-Improvement of Scene Script However, the agent often does not produce the correct layout outright. This is because either (a) the predicted constraints do not reflect the requirements in the input query or do not follow commonsense knowledge, which requires updating the scene graph edges $\mathcal{E}$ and the Blender code `code`); or (b) the generated constraint functions do not correctly reflect the semantic relationships and result in an incorrect layout (we need to update scoring functions $\mathcal{F}_r$ and arguments $\mathtt{arg}_r$).

We iteratively improve the initially generated scene layout in a visual feedback loop by taking advantage of the perception capabilities of a multimodal LLM (GPT-V (OpenAI, 2023)). We render the generated scene into an image, then feed the rendered image and the scene description directly to the `LLM+V-Reviewer`, asking it which constraints are lacking or not correctly satisfied, asking it to revise the script to reflect all the mistakes it finds. If `LLM+V-Reviewer` finds out that the error is rooted in the constraint functions, it can either modify existing functions or add new sub-functions to improve the layout planning for the current scene. This procedure repeats at every iteration of the feedback loop. We denote that at the t-th iteration, the function for each relation is $\mathcal{F}^t$ and the graph edges are $\mathcal{E}^t$, and the predicted optimal layout is $\hat{\mathcal{L}}^t$. This shares a similar intuition with recent works that utilize foundational models to generate a



```
  def parallelism_score(assets: List[Layout]) -> float:
      """Evaluates and returns a score indicating the degree of parallelism in a list of assets

      Args:
          assets: A list of asset layouts, each with a location attribute.

      Returns:
          A score between 0 and 1 indicating the parallelism of the assets.
      """
      # Calculate directional vectors between consecutive assets
      vectors = [
          calculate_vector(assets[i].location, assets[i + 1].location)
          for i in range(len(assets) - 1)
      ]

      # Normalize vectors to compare directions regardless of distance
      normalized_vectors = [normalize_vector(v) for v in vectors]

      # Calculate dot products between consecutive normalized vectors to evaluate parallelism
      dot_products = [
          np.dot(normalized_vectors[i], normalized_vectors[i + 1])
          for i in range(len(normalized_vectors) - 1)
      ]

- final_score = np.mean([0.5 * (dot + 1) for dot in dot_products])
+     # Rotational similarity
+     orientation_similarities = [
+         orientation_similarity(assets[i].orientation, assets[i + 1].orientation)
+         for i in range(len(assets) - 1)
+     ]
+
+     # Combine scores
+     position_score = np.mean([0.5 * (dot + 1) for dot in dot_products_position])
+     orientation_score = np.mean(
+         [(similarity + 1) / 2 for similarity in orientation_similarities]
+     )
+
+     # Average the position and orientation scores for the final score
+     final_score = (position_score + orientation_score) / 2
```

Figure 4. Example of function `parallelism_score` update in **outer-loop library learning** phase. The update adds constraint score forcing the orientation of the assets to be similar.

reward signal (Baumli et al., 2023; Rocamonde et al., 2023; Ma et al., 2023; Shinn et al., 2023). The feedback-loop optimization procedure can formally be written as:

$$\mathcal{E}^{(t+1)}, \mathcal{F}^{(t+1)}, \texttt{arg}^{(t+1)} \leftarrow \texttt{LLM+V-Reviewer}(\texttt{img}, q_k)$$

$$\text{subject to } \texttt{img} \leftarrow \texttt{Blender-Render}(\mathcal{A}, \mathcal{L}^t, \texttt{code}^t)$$

$$\mathcal{L}^t \leftarrow \underset{\mathcal{L}}{\text{argmax}} \sum_{r \in \mathcal{R}} F_r^t\big(\{\mathcal{L}(a_i) \mid a_i \in \mathcal{E}^t(r)\}, \texttt{arg}_r^t\big) \tag{5}$$

2.4. Library Learning

In the preceding sections, we described the methodology behind SceneCraft's generation of scenes, which involves the formulation of relations and constraint scoring functions, followed by their iterative optimization through a feedback loop. In this section, we go over the process by which we learn a comprehensive spatial skill library of constraint functions, designed for re-application in the scene generation process for new input queries.

Algorithm 1 Dual-Loop Improvement Workflow

```
Data: $L = \{F_r\}$: Initialize SceneCraft's library, $\mathcal{Q} = \{q\}$: a dataset of queries. $N_{\text{inner}}$ and $N_{\text{outer}}$: number of iterations for scene refinement and library learning.
repeat                                                    // outer-loop
    for $q \in \mathcal{Q}$ do
        Decompose $q$ into sub-scene descriptions and assets: $(q_1, \mathcal{A}_1) \ldots (q_K, \mathcal{A}_K) \leftarrow \texttt{LLM-decomposer}(q)$ for $k = 1$ to $K$ do
            Plan a scene graph $\mathcal{G}(s)$ for sub-scene $q_k$: $\mathcal{G}(s) \leftarrow \texttt{LLM-Planner}(q_k, \mathcal{A}_k)$
            Write code and arguments for scene $s_k$: $\texttt{code}, \texttt{arg} \leftarrow \texttt{LLM-Coder}(\mathcal{G}(s), q_k)$
            for $t = 1$ to $N_{inner}$ do                        // inner-loop
                Constraint-based Search: $\mathcal{L}^t \leftarrow \underset{\mathcal{L}}{\text{argmax}} \sum_{r \in \mathcal{R}} F_r^t\Big(\{\mathcal{L}(a_i) \mid a_i \in \mathcal{E}^t(r)\}, \texttt{arg}_r^t\Big)$
                Render: $\texttt{img} \leftarrow \texttt{Blender}(\mathcal{L}^t, \mathcal{A}, \texttt{code}^t)$
                Critic & Revise: $\mathcal{E}^{t+1}, \mathcal{F}^{t+1}, \texttt{arg}^{t+1} \leftarrow \texttt{LLM+V-Reviewer}(\texttt{img}, q_k)$
            end
        end
        Store function updates $F_r^T(q)$ for each relation $r$ per question $q$ into $\hat{F}_r$ set.
    end
    for $r \in \mathcal{R}$ do
        Update library $L$ with updated functions: $F_r \leftarrow \texttt{Library-Learner}\Big(\{\hat{F}_r(q) \mid q \in \mathcal{Q}\}\Big)$
    end
until $N_{outer}$ times;
```

The core of SceneCraft's library learning originates from the aforementioned self-refinement procedure. When a specific constraint function is not sufficient to cover all cases of a relation, the `LLM+V-Reviewer` is able to identify the pitfall of function implementation, and make corresponding modification. As an example shown in Figure 4, the previous implementation of *parallelism* relation only consider the assets' location. Through feedback-loop optimization for scene improvement, GPT-V identifies that it is necessary to consider the similarity over orientation. Therefore, the main goal of library learning procedure is to review these gradual changes of F_r, detect common patterns in the addition or modification, and merge these changes into the library.

Specifically, we denote $\hat{F}_r(q) = F_r^T(q)$ as the updated functions for relation r learned after $t = T$ step inner-loop self-improvement over query q. SceneCraft reviews all these updates, try to find one that represents the consensus of all, and merge it into the global skill library. This procedure shares similar intuition as universal self-consistency (Chen et al., 2023). We thus learn the new function as:

$$F_r \leftarrow \texttt{Library-Learner}\Big(\{\hat{F}_r(q) \mid q \in \mathcal{Q}\}\Big) \tag{6}$$

This process is conducted over a batch of queries $\mathcal{Q} = \{q_i\}$ to ensure the universality of the learned skills. Note that: 1) this procedure could be regarded as a meta-learning update of the function initialization to facilitate the feedback-loop optimization. 2) this procedure does not require any ground-truth scenes, explicit reward function, or any human intervention. All the internal learning signal is just the `LLM+V-reviewer` during the feedback-loop to maximize the alignment to textual query. 3) For both optimization stages, updates are made to non-parametric knowledge represented as Python code, avoiding the computational cost and inaccessibility issues associated with back-propagation in large language models.

SceneCraft's library learning process is also highly sample-efficient. By manually creating 20 examples with ground-truth constraints and running dual-stage optimization on them, SceneCraft develops a robust skill library. This approach contrasts with traditional model fine-tuning, offering efficiency and adaptability in learning for complex tasks like 3D scene generation. The pseudo-code of the whole dual-loop learning is illustrated in Alg 1.

3. Experiments

We evaluate our proposed SceneCraft first on our curated synthetic queries where the ground-truth constraints are available. We then show how the generated 3D scenes can help video generation on the Sintel movie as a case study.

3.1. Evaluate Scene Synthesis with Given Constraints

SceneCraft is an agent for open-domain scene synthesis. Most of the existing 3D scene datasets with ground-truth focus on a specific domain such as in-door scene (Song et al., 2023; Wei et al., 2023) or road traffic (Savkin et al., 2023). To systematically study and evaluate our agent in this task, we manually create 40 synthetic queries with ground-truth constraints. The way we generate these queries is by first sampling a subset of relation constraints from the full list (shown in Appendix C). Based on this, the human annotators evaluate whether the scene satisfies this relational constraint. Assets are retrieved from Turbosquid[2].

Evaluation Metric To verify whether a generated scene fulfills the textual requirement, we ask human annotators to also write a scoring function to estimate how much the constraint is satisfied. Such function is different from the one SceneCraft learns in its skill library, because the scoring function only needs to work for this specific scene query. We show examples of the queries as well as the implemented scoring function in Sec F in Appendix. The output of these scoring functions is less than or equal to 1, and only reaches equality when all constraints are strictly satisfied.

For this synthetic dataset, as we don't have the ground-truth scene layout, we adopt two metrics for evaluating scene synthesis model's performance. The first is the standard text-to-image CLIP similarity score (Radford et al., 2021), which measures how well the generated scene satisfies the textual description; we also use the functions human annotators wrote as a more fine-grained evaluation on how our generated scene satisfies all the semantic requirements hidden in the query. We use 20 of the 40 queries for building the spatial skill library through dual-loop optimization, during which the model only sees the query instead of ground-truth constraint score. Afterwards, we evaluate the model performance on the remaining 20 queries.

Baselines Most of the existing 3D scene synthesis works only focus on a specific domain, e.g., indoor scenes. The only prior system that serves similar purpose to SceneCraft might be BlenderGPT[3], an LLM assistant that also takes text query as input and generates Blender code. The main difference of BlenderGPT against SceneCraft is that BlenderGPT is limited to only the basic Blender instructions such as moving an asset or changing texture. To allow it to solve the text-to-scene synthesis task, we modify their code to: 1) enable BlenderGPT to use GPT-V to receive the screenshot of Blender as visual feedback; 2) asking itself to give the per-step instruction for generating the complex scene. We also report results of our own system's ablation. There are three major design choices of SceneCraft: 1) Abstraction of scene as relational graph; 2) inner-loop optimization of the scene with visual feedback; 3) outer-loop learning of skill library. These three components have some dependencies: constraint function grounded by relational graph is the main interface to be updated by inner-loop (BlenderGPT can be regarded as a baseline only with inner-loop update but without graph grounding); while the inner-loop updates of function is the root for library learning. Therefore, we do ablation study by removing one component after the other.

Experimental Results Results are shown in Table 1. We see that our method consistently improves over all baselines in terms of both CLIP similarity as well as the constraint score. Notably, on the constraint score, the BlenderGPT baseline only achieves 5.6 score. We show a few head-to-head comparisons in Figure 1. As example, in the first query that asks three boxes stack one on top of each other, BlenderGPT simply lists the three boxes in a line and does not follow the instruction of stacking; on the second query that asks three trees in a row, BlenderGPT does organize the trees in a line, but perpendicular with the road edge. These examples show that BlenderGPT without the relational constraint is not able to conduct complex spatial planning.

In the meantime, the ablation studies by removing each component also shows that all components are very crucial for Scenecraft. Among these components, inner-loop optimization provides the most important leaps; removing it leads to 38.4 drop on constraint score, and it's also the root for library learning that keep the system self-improving without human annotation.

We also conduct a qualitative evaluation of SceneCraft's

[2] https://www.turbosquid.com/

[3] https://github.com/gd3kr/BlenderGPT

Metric	CLIP SIM	Constraint Score
BlenderGPT	24.7	5.6
SceneCraft	**69.8**	**88.9**
(—-Ablation by removing one component after the other—-)		
– Learned Library	48.3	64.5
– Inner-Loop	32.8	26.1
– Relation Graph	19.4	3.2

Table 1. Comparison of SceneCraft against BlenderGPT and ablation baselines on synthetic queries with annotated constraints.

Win Rate	Text Fidelity	Composition	Aesthetics
SceneCraft	76.8%	83.6%	74.5%
BlenderGPT	12.7%	11.4%	14.5%

Table 2. Qualitative Human comparison of SceneCraft against BlenderGPT baseline.

output versus BlenderGPT baseline. We randomly select 10 pairs generated by SceneCraft and BlenderGPT, and ask humans to judge which one is better, in terms of three major dimensions: 1) text fidelity: how much the generated scene aligns with the textual query; 2) composition & constraint agreement: we tell the raters the ground-truth relations, and ask whether the generated scene follows all these constraints; 3) Aesthetics: we ask which output has better overall visual quality. The order of our output against baseline is completely random. Detailed question and interface is shown in Figure 14 in Appendix. Altogether we collect 22 responses. Results in Table 2 show that our method outperforms BlenderGPT in all the three dimensions significantly. Specifically, consistent with our results on constraint score, SceneCraft gains more improvement over the constraint agreement, making the scene logically correct.

3.2. Scene-Guided Video Generation over Sintel Movie

In addition to synthetic queries, we also show that SceneCraft's layout planning capability generalizes to real scenes , and has potential to control and benefit video generation. As open-domain videos do not always have ground-truth scenes, we take the Sintel Movie, which is an animated fantasy short film produced with Blender, where scripts and Blender scenes are open sourced[4]. We download all these scenes, using the first half as the training set and the remaining half for testing. For this task, we assume that the model is given the ground-truth assets for the scene, and only focuses on layout planning to satisfy the textual description. After we recover the scene, we study how it can benefit a video generation model to get higher-quality predictions. We thus fine-tune the VideoPoet model (Kondratyuk et al., 2023), an autoregressive Transformer-based video generation framework, on the training set with one ground-truth scene image frame as a conditional input. The image will be converted into image tokens, and add as prefix after text prompt. We then take the fine-tuned VideoPoet model, taking our model and BlenderGPT's predicted scene, to generate a 2 second video.

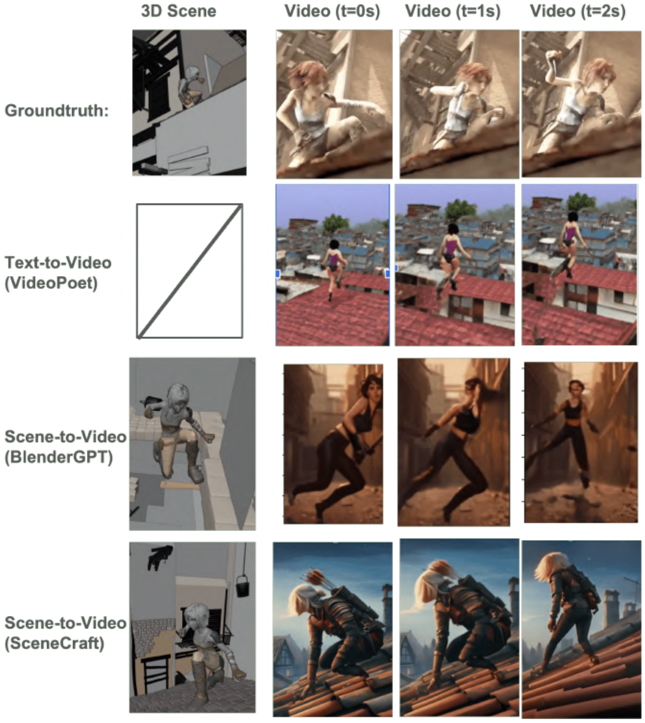

3D Scene
Video (t=0s)
Video (t=1s)
Video (t=2s)
Groundtruth:
Text-to-Video (VideoPoet)
Scene-to-Video (BlenderGPT)
Scene-to-Video (SceneCraft)

Figure 5. Predicted 3D Scenes as well as the generated videos by SceneCraft against other baselines.

We compare the output in terms of both the scene itself as well as how much it benefits the overall video generation. For the scene, we use two metrics: the layout matrix's similarity (first calculate mutual similarity between assets, then calculate cosine similarity), and the rendered image's CLIP score. For the video, as we use both the standard Frechet Video Distance (FVD) distribution score (Unterthiner et al., 2019), as well as CLIP-based Relative Matching (RM) score (Wu et al., 2021). The results shown in Table 3 illustrate that our method consistently improves the BlenderGPT output in terms of scene planning. In addition, the generated scene helps the video generation and outperform the vanilla text-to-video baseline. From the examples in Figure 5, we can see that the 3D scene grounding help the generated video follow more similar structure as ground-truth ones. This shows the potential of SceneCraft in controlling video generation in wider domain.

4. Related Works

Text to 3D-Scene Synthesis One of the earliest forays into text-driven 3D scene synthesis is WordsEye (Coyne & Sproat, 2001). This system, and its follow-up works (Seversky & Yin, 2006; Chang et al., 2014; Ma et al., 2018), can

[4] https://studio.blender.org/films/sintel/

Quantitative Metric	Scene Comparison		Video Comparison	
	Layout Matrix SIM	**Scene CLIP SIM**	**CLIP-based RM**	**FVD ↓**
Text-to-Video (w.o. / finetune)	/	/	56.8	846
Text-to-Video (w / finetune)	/	/	64.2	531
Text-to-Scene-to-Video, finetune a videogen model on groundtruth scene, infer with scenes generated by:				
BlenderGPT	27.5	41.8	69.1	574
SceneCraft (Dual-Loop)	69.3	82.7	46.2	317

Table 3. Comparison of SceneCraft with other ablated baselines on a Sintel movie. In this setting, we assume to be given fixed assets for each scene, try to recover the scene, and guide a video generative model which is fine-tuned on first half of the video. We compare with naiive text-to-video baselines without scene guidance.

generate 3D scenes from natural language. However, these systems often require manual mapping between language and object placement, leading to somewhat unnatural commands for scene description. Zitnick et al. (2013) learns to map visual features to semantic phrases extracted from sentences, focusing on binary spatial or semantic relationships. Chang et al. (2014) build upon and improve these early systems. The key advancement is the use of spatial knowledge, derived from 3D scene data, to more accurately constrain scene generations. This approach allows for a more realistic interpretation of unstated facts or common sense in scene synthesis. In their subsequent work (Chang et al., 2015), they focused on lexical grounding of textual terms to 3D model references, combining rule-based models with user annotations to select appropriate objects. Their latest paper (Chang et al., 2017) further refines this approach, introducing interactive text-based scene editing operations and an improved user interface. All these systems are most purely symbolic rule-based and require significant human efforts to maintain, and are, therefore, hard to generalize to new domains and types of constraints.

There also exist a line of neural-based 3D scene generation that learns from data. Most works in this direction focus on a specific domain, such as in-door scenes (Patil et al., 2023). For instance, RoomDreamer (Song et al., 2023) trains a diffusion model to simultaneously generate layout, geometry and texture for in-door scenes; LEGO-Net (Wei et al., 2023) focus on the layout planning, and trains a Transformer model to iteratively cleanup the messy room. Despite the impressive performance of these work, they are restricted by the available 3D scene data. For most open-domain image and videos, it is very hard to collect ground-truth 3D scenes, which is why most works in this domain focus on in-door scenes. On the contrary, this paper focus on exploring whether we can take advantage of the existing knowledge and reasoning capabilities of Large Language Models to directly do layout planning without tuning its parameters, and we try to learn general spatial planning skills that can be generalized from very small number of synthetic queries.

Multimodal LLM Agents Leverageing visual perception abilities of recent models like GPT-V, multimodal LLM Agents (Liu et al., 2023) are capable of interacting with external visual environments, such as web browsing (Deng et al., 2023; Zhou et al., 2023b; Hu et al., 2023; Zheng et al., 2024), gaming (Wang et al., 2023), robotics (Brohan et al., 2023) and design (Lv et al., 2023; Yang et al., 2024). The most related concurrent works is 3D-GPT (Sun et al., 2023), which interacts with Infinigen (Raistrick et al., 2023), a high-level wrapper on top of Blender, to create high-quality environmental scenes. The main difference of our work against 3D-GPT is: 1) Environment-wise, we directly interact with Blender and a large-scale asset pool, which provide richer assets to construct the scene, while Infinigen for now only supports limited number of assets and environment arguments; 2) methodology-wise, SceneCraft features a dual-loop self-improvement pipeline, which enables us to learn new design skills to handle unseen tasks, which differentiate us to many existing llm-agent works that heavily rely on manual prompt design.

5. Conclusion

In this paper, we present SceneCraft, an LLM-powered autonomous agent for transforming input text query to a 3D Scene by generating a Blender-executable Python script. Scenecraft builds on top of multimodal LLMs for both planning and library learning in a dual-loop self-improving framework. In the inner-loop, SceneCraft generates Blender-executable Python scripts to render an image of the scene, and use a Self-critiquing loop to iteratively refine its output and learn from its performance. The outer-loop dynamically expands a 'spatial skill' library, facilitating continuous self-improvement without the need for expensive LLM parameter tuning. In the future, we'd like to explore: 1) using our framework for reconstructing the 3D scene corresponding to a given open-domain image or video; 2) utilizing the generated dataset to fine-tune a video generation conditioned on a 3D scene as control signal.

Impact Statement

This paper presents work whose goal is to advance the text-to-3d-scene synthesis. The work can have potential to benefit the gaming, cinematic and design industry, which are mostly positive impact. We only learn a non-parametric skill library from synthetic queries, and not using any private information. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Blender. https://www.blender.org/.

txtai: the all-in-one embeddings database. https://github.com/neuml/txtai.

Baumli, K., Baveja, S., Behbahani, F. M. P., Chan, H., Comanici, G., Flennerhag, S., Gazeau, M., Holsheimer, K., Horgan, D., Laskin, M., Lyle, C., Masoom, H., McKinney, K., Mnih, V., Neitz, A., Pardo, F., Parker-Holder, J., Quan, J., Rocktäschel, T., Sahni, H., Schaul, T., Schroecker, Y., Spencer, S., Steigerwald, R., Wang, L., and Zhang, L. Vision-language models as a source of rewards. *CoRR*, abs/2312.09187, 2023. doi: 10.48550/ARXIV.2312.09187. URL https://doi.org/10.48550/arXiv.2312.09187.

Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Chen, X., Choromanski, K., Ding, T., Driess, D., Dubey, A., Finn, C., Florence, P., Fu, C., Arenas, M. G., Gopalakrishnan, K., Han, K., Hausman, K., Herzog, A., Hsu, J., Ichter, B., Irpan, A., Joshi, N. J., Julian, R., Kalashnikov, D., Kuang, Y., Leal, I., Lee, L., Lee, T. E., Levine, S., Lu, Y., Michalewski, H., Mordatch, I., Pertsch, K., Rao, K., Reymann, K., Ryoo, M. S., Salazar, G., Sanketi, P., Sermanet, P., Singh, J., Singh, A., Soricut, R., Tran, H. T., Vanhoucke, V., Vuong, Q., Wahid, A., Welker, S., Wohlhart, P., Wu, J., Xia, F., Xiao, T., Xu, P., Xu, S., Yu, T., and Zitkovich, B. RT-2: vision-language-action models transfer web knowledge to robotic control. *CoRR*, abs/2307.15818, 2023. doi: 10.48550/ARXIV.2307.15818. URL https://doi.org/10.48550/arXiv.2307.15818.

Chang, A. X., Savva, M., and Manning, C. D. Learning spatial knowledge for text to 3d scene generation. In Moschitti, A., Pang, B., and Daelemans, W. (eds.), *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pp. 2028–2038. ACL, 2014. doi: 10.3115/V1/D14-1217. URL https://doi.org/10.3115/v1/d14-1217.

Chang, A. X., Monroe, W., Savva, M., Potts, C., and Manning, C. D. Text to 3d scene generation with rich lexical grounding. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pp. 53–62. The Association for Computer Linguistics, 2015. doi: 10.3115/V1/P15-1006. URL https://doi.org/10.3115/v1/p15-1006.

Chang, A. X., Eric, M., Savva, M., and Manning, C. D. Sceneseer: 3d scene design with natural language. *CoRR*, abs/1703.00050, 2017. URL http://arxiv.org/abs/1703.00050.

Chen, X., Aksitov, R., Alon, U., Ren, J., Xiao, K., Yin, P., Prakash, S., Sutton, C., Wang, X., and Zhou, D. Universal self-consistency for large language model generation. *CoRR*, abs/2311.17311, 2023. doi: 10.48550/ARXIV.2311.17311. URL https://doi.org/10.48550/arXiv.2311.17311.

Coyne, R. and Sproat, R. Wordseye: an automatic text-to-scene conversion system. In Pocock, L. (ed.), *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001, Los Angeles, California, USA, August 12-17, 2001*, pp. 487–496. ACM, 2001. doi: 10.1145/383259.383316. URL https://doi.org/10.1145/383259.383316.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. Mind2web: Towards a generalist agent for the web. *CoRR*, abs/2306.06070, 2023. doi: 10.48550/ARXIV.2306.06070. URL https://doi.org/10.48550/arXiv.2306.06070.

Epstein, D., Poole, B., Mildenhall, B., Efros, A. A., and Holynski, A. Disentangled 3d scene generation with layout learning. *CoRR*, abs/2402.16936, 2024. doi: 10.48550/ARXIV.2402.16936. URL https://doi.org/10.48550/arXiv.2402.16936.

Gao, G., Liu, W., Chen, A., Geiger, A., and Schölkopf, B. Graphdreamer: Compositional 3d scene synthesis from scene graphs. *CoRR*, abs/2312.00093, 2023. doi: 10.48550/ARXIV.2312.00093. URL https://doi.org/10.48550/arXiv.2312.00093.

Hu, Z., Iscen, A., Sun, C., Chang, K., Sun, Y., Ross, D. A., Schmid, C., and Fathi, A. AVIS: autonomous visual information seeking with large language models. *CoRR*, abs/2306.08129, 2023. doi: 10.48550/ARXIV.2306.08129. URL https://doi.org/10.48550/arXiv.2306.08129.

Kondratyuk, D., Yu, L., Gu, X., Lezama, J., Huang, J., Hornung, R., Adam, H., Akbari, H., Alon, Y., Birodkar, V., Cheng, Y., Chiu, M., Dillon, J., Essa, I., Gupta, A., Hahn, M., Hauth, A., Hendon, D., Martinez, A., Minnen, D., Ross, D. A., Schindler, G., Sirotenko, M., Sohn, K., Somandepalli, K., Wang, H., Yan, J., Yang, M., Yang, X., Seybold, B., and Jiang, L. Videopoet: A large language model for zero-shot video generation. *CoRR*, abs/2312.14125, 2023. doi: 10.48550/ARXIV.2312.14125. URL https://doi.org/10.48550/arXiv.2312.14125.

Lin, C., Gao, J., Tang, L., Takikawa, T., Zeng, X., Huang, X., Kreis, K., Fidler, S., Liu, M., and Lin, T. Magic3d: High-resolution text-to-3d content creation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pp. 300–309. IEEE, 2023. doi: 10.1109/CVPR52729.2023.00037. URL https://doi.org/10.1109/CVPR52729.2023.00037.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. Agentbench: Evaluating llms as agents. *CoRR*, abs/2308.03688, 2023. doi: 10.48550/ARXIV.2308.03688. URL https://doi.org/10.48550/arXiv.2308.03688.

Lv, J., Huang, Y., Yan, M., Huang, J., Liu, J., Liu, Y., Wen, Y., Chen, X., and Chen, S. Gpt4motion: Scripting physical motions in text-to-video generation via blender-oriented gpt planning. *arXiv preprint arXiv:2311.12631*, 2023.

Ma, R., Patil, A. G., Fisher, M., Li, M., Pirk, S., Hua, B., Yeung, S., Tong, X., Guibas, L. J., and Zhang, H. Language-driven synthesis of 3d scenes from scene databases. *ACM Trans. Graph.*, 37(6):212, 2018. doi: 10.1145/3272127.3275035. URL https://doi.org/10.1145/3272127.3275035.

Ma, Y. J., Liang, W., Wang, G., Huang, D., Bastani, O., Jayaraman, D., Zhu, Y., Fan, L., and Anandkumar, A. Eureka: Human-level reward design via coding large language models. *CoRR*, abs/2310.12931, 2023. doi: 10.48550/ARXIV.2310.12931. URL https://doi.org/10.48550/arXiv.2310.12931.

OpenAI. Gpt-4v(ision) system card. System Card, 2023. URL URL_of_the_System_Card. Version 1.0.

Patil, A. G., Patil, S. G., Li, M., Fisher, M., Savva, M., and Zhang, H. Advances in data-driven analysis and synthesis of 3d indoor scenes. *CoRR*, abs/2304.03188, 2023. doi: 10.48550/ARXIV.2304.03188. URL https://doi.org/10.48550/arXiv.2304.03188.

Perez, E., Lewis, P. S. H., Yih, W., Cho, K., and Kiela, D. Unsupervised question decomposition for question answering. In Webber, B., Cohn, T., He, Y., and Liu, Y. (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pp. 8864–8880. Association for Computational Linguistics, 2020. doi: 10.18653/V1/2020.EMNLP-MAIN.713. URL https://doi.org/10.18653/v1/2020.emnlp-main.713.

Poole, B., Jain, A., Barron, J. T., and Mildenhall, B. Dreamfusion: Text-to-3d using 2d diffusion. *CoRR*, abs/2209.14988, 2022. doi: 10.48550/ARXIV.2209.14988. URL https://doi.org/10.48550/arXiv.2209.14988.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 2021. URL http://proceedings.mlr.press/v139/radford21a.html.

Raistrick, A., Lipson, L., Ma, Z., Mei, L., Wang, M., Zuo, Y., Kayan, K., Wen, H., Han, B., Wang, Y., Newell, A., Law, H., Goyal, A., Yang, K., and Deng, J. Infinite photorealistic worlds using procedural generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pp. 12630–12641. IEEE, 2023. doi: 10.1109/CVPR52729.2023.01215. URL https://doi.org/10.1109/CVPR52729.2023.01215.

Rocamonde, J., Montesinos, V., Nava, E., Perez, E., and Lindner, D. Vision-language models are zero-shot reward models for reinforcement learning. *CoRR*, abs/2310.12921, 2023. doi: 10.48550/ARXIV.2310.12921. URL https://doi.org/10.48550/arXiv.2310.12921.

Savkin, A., Ellouze, R., Navab, N., and Tombari, F. Unsupervised traffic scene generation with synthetic 3d scene graphs. *CoRR*, abs/2303.08473, 2023. doi: 10.48550/ARXIV.2303.08473. URL https://doi.org/10.48550/arXiv.2303.08473.

Seversky, L. M. and Yin, L. Real-time automatic 3d scene generation from natural language voice and text descriptions. In Nahrstedt, K., Turk, M. A., Rui, Y., Klas, W., and Mayer-Patel, K. (eds.), *Proceedings of the 14th ACM International Conference on Multimedia, Santa Barbara, CA, USA, October 23-27, 2006*, pp. 61–64. ACM,

2006. doi: 10.1145/1180639.1180660. URL https://doi.org/10.1145/1180639.1180660.

Shinn, N., Labash, B., and Gopinath, A. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

Song, L., Cao, L., Xu, H., Kang, K., Tang, F., Yuan, J., and Yang, Z. Roomdreamer: Text-driven 3d indoor scene synthesis with coherent geometry and texture. In El-Saddik, A., Mei, T., Cucchiara, R., Bertini, M., Vallejo, D. P. T., Atrey, P. K., and Hossain, M. S. (eds.), *Proceedings of the 31st ACM International Conference on Multimedia, MM 2023, Ottawa, ON, Canada, 29 October 2023- 3 November 2023*, pp. 6898–6906. ACM, 2023. doi: 10.1145/3581783.3611800. URL https://doi.org/10.1145/3581783.3611800.

Sun, C., Han, J., Deng, W., Wang, X., Qin, Z., and Gould, S. 3d-gpt: Procedural 3d modeling with large language models. *CoRR*, abs/2310.12945, 2023. doi: 10.48550/ARXIV.2310.12945. URL https://doi.org/10.48550/arXiv.2310.12945.

Tang, J., Ren, J., Zhou, H., Liu, Z., and Zeng, G. Dreamgaussian: Generative gaussian splatting for efficient 3d content creation. *CoRR*, abs/2309.16653, 2023. doi: 10.48550/ARXIV.2309.16653. URL https://doi.org/10.48550/arXiv.2309.16653.

Unterthiner, T., van Steenkiste, S., Kurach, K., Marinier, R., Michalski, M., and Gelly, S. FVD: A new metric for video generation. In *Deep Generative Models for Highly Structured Data, ICLR 2019 Workshop, New Orleans, Louisiana, United States, May 6, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=rylgEULtdN.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *CoRR*, abs/2305.16291, 2023. doi: 10.48550/ARXIV.2305.16291. URL https://doi.org/10.48550/arXiv.2305.16291.

Wei, Q. A., Ding, S., Park, J. J., Sajnani, R., Poulenard, A., Sridhar, S., and Guibas, L. J. Lego-net: Learning regular rearrangements of objects in rooms. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pp. 19037–19047. IEEE, 2023. doi: 10.1109/CVPR52729.2023.01825. URL https://doi.org/10.1109/CVPR52729.2023.01825.

Wen, Z., Liu, Z., Sridhar, S., and Fu, R. Anyhome: Open-vocabulary generation of structured and textured 3d homes. *CoRR*, abs/2312.06644, 2023. doi: 10.48550/ARXIV.2312.06644. URL https://doi.org/10.48550/arXiv.2312.06644.

Wu, C., Huang, L., Zhang, Q., Li, B., Ji, L., Yang, F., Sapiro, G., and Duan, N. GODIVA: generating open-domain videos from natural descriptions. *CoRR*, abs/2104.14806, 2021. URL https://arxiv.org/abs/2104.14806.

Yang, L., Yu, Z., Meng, C., Xu, M., Ermon, S., and Cui, B. Mastering text-to-image diffusion: Recaptioning, planning, and generating with multimodal llms. *arXiv preprint arXiv:2401.11708*, 2024.

Yang, Y., Sun, F., Weihs, L., VanderBilt, E., Herrasti, A., Han, W., Wu, J., Haber, N., Krishna, R., Liu, L., Callison-Burch, C., Yatskar, M., Kembhavi, A., and Clark, C. Holodeck: Language guided generation of 3d embodied AI environments. *CoRR*, abs/2312.09067, 2023. doi: 10.48550/ARXIV.2312.09067. URL https://doi.org/10.48550/arXiv.2312.09067.

Zelikman, E., Huang, Q., Poesia, G., Goodman, N. D., and Haber, N. Parsel: Algorithmic reasoning with language models by composing decompositions. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Zhang, Q., Wang, C., Siarohin, A., Zhuang, P., Xu, Y., Yang, C., Lin, D., Zhou, B., Tulyakov, S., and Lee, H. Scenewiz3d: Towards text-guided 3d scene composition. *CoRR*, abs/2312.08885, 2023. doi: 10.48550/ARXIV.2312.08885. URL https://doi.org/10.48550/arXiv.2312.08885.

Zheng, B., Gou, B., Kil, J., Sun, H., and Su, Y. Gpt-4v(ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q. V., and Chi, E. H. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023a. URL https://openreview.net/pdf?id=WZH7099tgfM.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Bisk, Y., Fried, D., Alon, U., et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023b. URL https://webarena.dev.

Zitnick, C. L., Parikh, D., and Vanderwende, L. Learning the visual interpretation of sentences. In *IEEE International Conference on Computer Vision, ICCV 2013, Sydney, Australia, December 1-8, 2013*, pp. 1681–1688. IEEE Computer Society, 2013. doi: 10.1109/ICCV.2013.211. URL https://doi.org/10.1109/ICCV.2013.211.

Supplementary Material for SCENECRAFT

A. Discussion with other Related Works

Text-to-3D objects including (Poole et al., 2022) and (Tang et al., 2023) focus on using generative models to generate a single asset/object, consisting of its shape, texture, geometry. Recent efforts enable textual control to describe the asset's properties to provide better control for each single asset. SceneCraft tries to tackle the layout planning problem on top of a single asset and generates a scene consisting of multiple assets by correctly placing each asset to a reasonable location. With the integration of low-level single-asset generation and scenecraft, we provide a more controllable 3D modeling interface (e.g., we can also ask the LLM to prepare a textual prompt to feed into this single-asset text-to-3D model to give us the customized asset, and we focus on planning, and the LLM Agent can focus on deciding which texture and which shape shall this asset have).

There exist several concurrent works also on scene layout planning. For example (Epstein et al., 2024), (Zhang et al., 2023) and (Yang et al., 2023) all try to use LLM to synthesize constraint-based scene graphs and optimize the layout with human-defined constraint functions. The main difference of SceneCraft is that we use a self-improvement loop to enable LLM itself to learn and maintain the design library. (Gao et al., 2023) is a very interesting work to compose multiple single-object nerf to do text-to-scene generation. Scene generation is done purely in neural embedding space, with the hypothesis that the scene can be decomposed into disentangled objects. (Wen et al., 2023) also tries to combine multiple single-asset nerf representation to compose the global scene, and they use a diffusion model as differentiable rendering to receive guidance from text-image alignment and backpropagate to optimize layout. This shares similar intuition of our GPT4-V critic and revise, while the main difference is that we perform non-parametric update that relies on LLM to directly manipulate the code and symbol, while (Wen et al., 2023) is mainly conducting in embedding space.

B. Examples of SceneCraft's Generated Scripts and Rendered Scenes

Example of SceneCraft's generated scripts and rendered scene on the synthetic datasets are in Figure 6 and Figure 7.

C. List of relationships

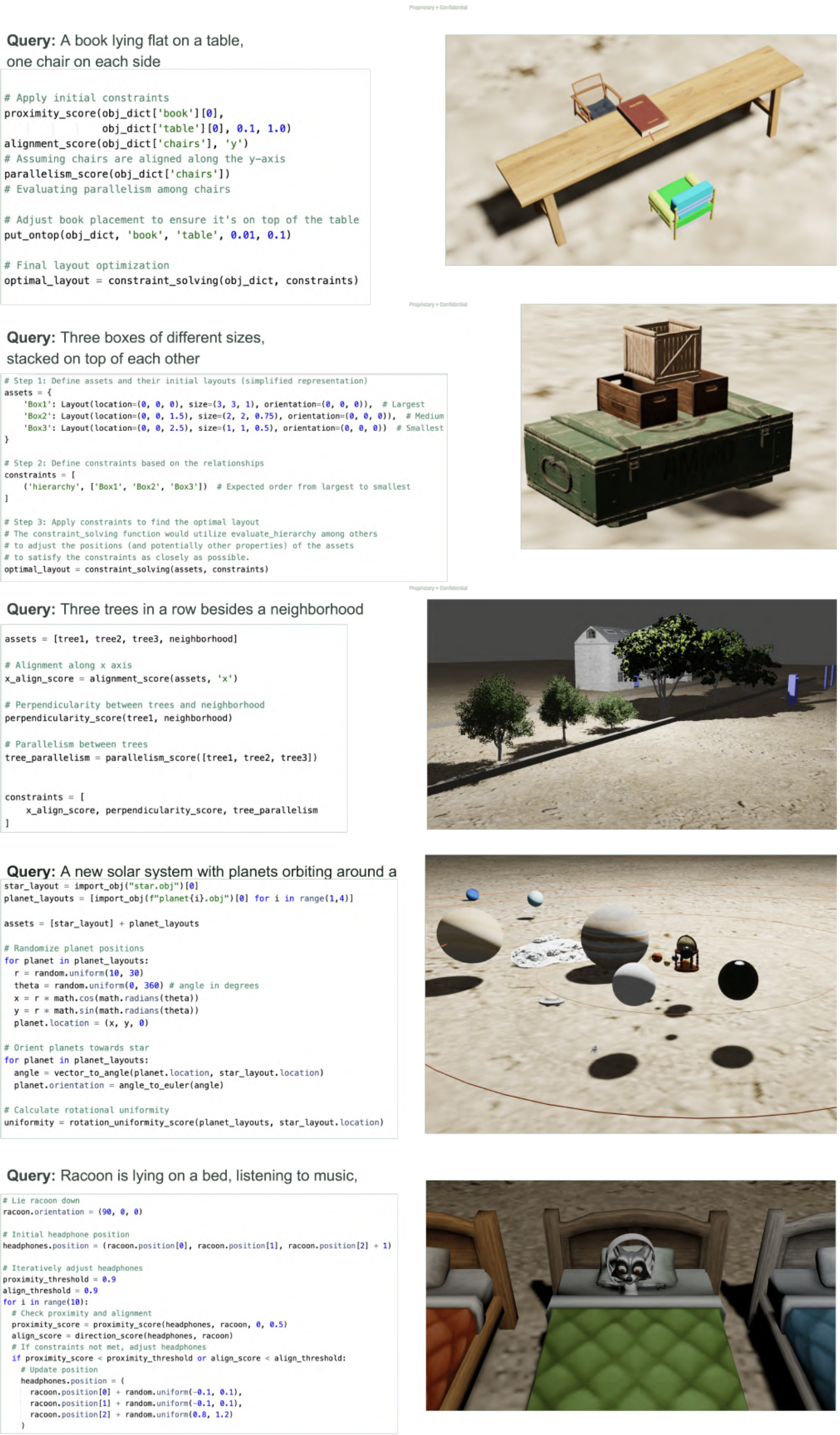
SceneCraft encapsulates several types of relationships and constraints, including:

- **Proximity**: A constraint enforcing the closeness of two objects, e.g., a chair near a table.
- **Direction**: The angle of one object is targeting at the other.
- **Alignment**: Ensuring objects align along a common axis, e.g., paintings aligned vertically on a wall.
- **Symmetry**: Mirroring objects along an axis, e.g., symmetrical placement of lamps on either side of a bed.
- **Overlap**: One object partially covering another, creating depth, e.g., a rug under a coffee table.
- **Parallelism**: Objects parallel to each other, suggesting direction, e.g., parallel rows of seats in a theater.
- **Perpendicularity**: Objects intersecting at a right angle, e.g., a bookshelf perpendicular to a desk.
- **Hierarchy**: Indicating a list of objects follow a certain order of size / volumns.
- **Rotation**: a list of objects rotate a cirtain point, e.g., rotating chairs around a meeting table.
- **Repetition**: Repeating patterns for rhythm or emphasis, e.g., a sequence of street lights.
- **Scaling**: Adjusting object sizes for depth or focus, e.g., smaller background trees to create depth perception.

These relationships are vital for creating scenes that are not only visually appealing but also contextually coherent. Traditionally the functions $F(\cdot)$ for each constraint shall be written by human experts, and SceneCraft's major contribution is to autonomously learn and evolve the library of constraint satisfaction functions $\mathcal{F} = \{F_c(\cdot)\}_{c \in C}$, using a Large Language Model (LLM) Agent.

D. Spatial Skill Library

Below listed all the functions our framework generate. There exist some basic fundamental editing functions like import object, add camera, scaling, repetition; some functions to get information from the scene, such as calculate shortest distance between objects, calculate volumn, etc; as well as functions that calculate constraint satisfying score for each relationship. All these functions are autonomously written and modified by LLM Agent itself, without ground-truth label or explicit



Query: A book lying flat on a table, one chair on each side

```
# Apply initial constraints
proximity_score(obj_dict['book'][0],
                obj_dict['table'][0], 0.1, 1.0)
alignment_score(obj_dict['chairs'], 'y')
# Assuming chairs are aligned along the y-axis
parallelism_score(obj_dict['chairs'])
# Evaluating parallelism among chairs

# Adjust book placement to ensure it's on top of the table
put_ontop(obj_dict, 'book', 'table', 0.01, 0.1)

# Final layout optimization
optimal_layout = constraint_solving(obj_dict, constraints)
```

Query: Three boxes of different sizes, stacked on top of each other

```
# Step 1: Define assets and their initial layouts (simplified representation)
assets = {
    'Box1': Layout(location=(0, 0, 0), size=(3, 3, 1), orientation=(0, 0, 0)),  # Largest
    'Box2': Layout(location=(0, 0, 1.5), size=(2, 2, 0.75), orientation=(0, 0, 0)),  # Medium
    'Box3': Layout(location=(0, 0, 2.5), size=(1, 1, 0.5), orientation=(0, 0, 0))  # Smallest
}

# Step 2: Define constraints based on the relationships
constraints = [
    ('hierarchy', ['Box1', 'Box2', 'Box3'])  # Expected order from largest to smallest
]

# Step 3: Apply constraints to find the optimal layout
# The constraint_solving function would utilize evaluate_hierarchy among others
# to adjust the positions (and potentially other properties) of the assets
# to satisfy the constraints as closely as possible.
optimal_layout = constraint_solving(assets, constraints)
```

Query: Three trees in a row besides a neighborhood

```
assets = [tree1, tree2, tree3, neighborhood]

# Alignment along x axis
x_align_score = alignment_score(assets, 'x')

# Perpendicularity between trees and neighborhood
perpendicularity_score(tree1, neighborhood)

# Parallelism between trees
tree_parallelism = parallelism_score([tree1, tree2, tree3])


constraints = [
    x_align_score, perpendicularity_score, tree_parallelism
]
```

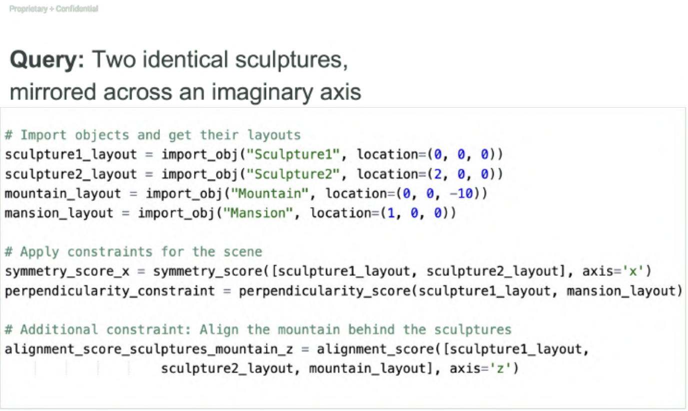
Query: A new solar system with planets orbiting around a

```
star_layout = import_obj("star.obj")[0]
planet_layouts = [import_obj(f"planet{i}.obj")[0] for i in range(1,4)]

assets = [star_layout] + planet_layouts

# Randomize planet positions
for planet in planet_layouts:
  r = random.uniform(10, 30)
  theta = random.uniform(0, 360) # angle in degrees
  x = r * math.cos(math.radians(theta))
  y = r * math.sin(math.radians(theta))
  planet.location = (x, y, 0)

# Orient planets towards star
for planet in planet_layouts:
  angle = vector_to_angle(planet.location, star_layout.location)
  planet.orientation = angle_to_euler(angle)

# Calculate rotational uniformity
uniformity = rotation_uniformity_score(planet_layouts, star_layout.location)
```

Query: Racoon is lying on a bed, listening to music,

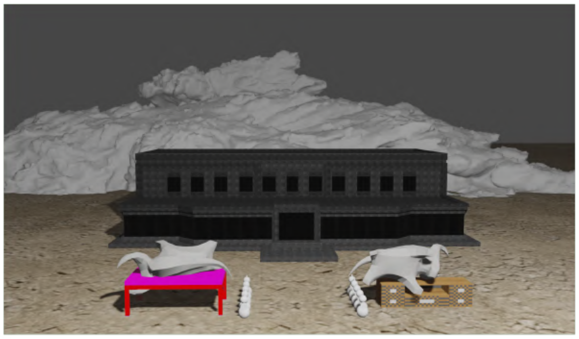
```
# Lie racoon down
racoon.orientation = (90, 0, 0)

# Initial headphone position
headphones.position = (racoon.position[0], racoon.position[1], racoon.position[2] + 1)

# Iteratively adjust headphones
proximity_threshold = 0.9
align_threshold = 0.9
for i in range(10):
  # Check proximity and alignment
  proximity_score = proximity_score(headphones, racoon, 0, 0.5)
  align_score = direction_score(headphones, racoon)
  # If constraints not met, adjust headphones
  if proximity_score < proximity_threshold or align_score < align_threshold:
    # Update position
    headphones.position = (
      racoon.position[0] + random.uniform(-0.1, 0.1),
      racoon.position[1] + random.uniform(-0.1, 0.1),
      racoon.position[2] + random.uniform(0.8, 1.2)
    )
```

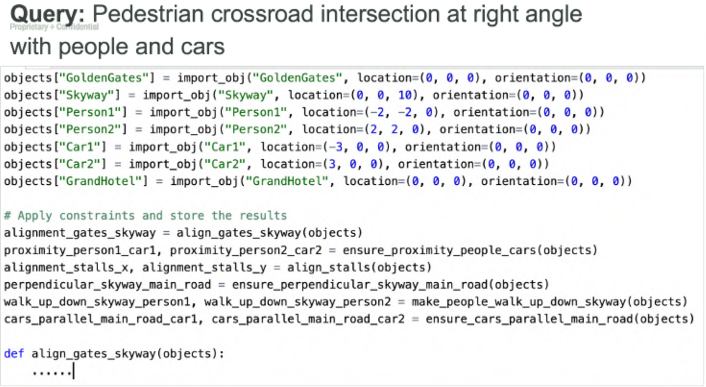
Figure 6. Examples of generated code and scenes



Query: Two identical sculptures, mirrored across an imaginary axis

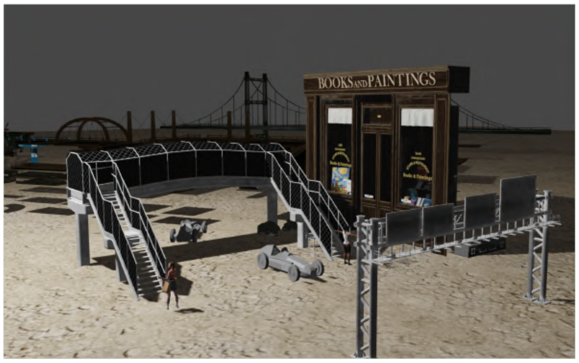
```
# Import objects and get their layouts
sculpture1_layout = import_obj("Sculpture1", location=(0, 0, 0))
sculpture2_layout = import_obj("Sculpture2", location=(2, 0, 0))
mountain_layout = import_obj("Mountain", location=(0, 0, -10))
mansion_layout = import_obj("Mansion", location=(1, 0, 0))

# Apply constraints for the scene
symmetry_score_x = symmetry_score([sculpture1_layout, sculpture2_layout], axis='x')
perpendicularity_constraint = perpendicularity_score(sculpture1_layout, mansion_layout)

# Additional constraint: Align the mountain behind the sculptures
alignment_score_sculptures_mountain_z = alignment_score([sculpture1_layout,
                    sculpture2_layout, mountain_layout], axis='z')
```

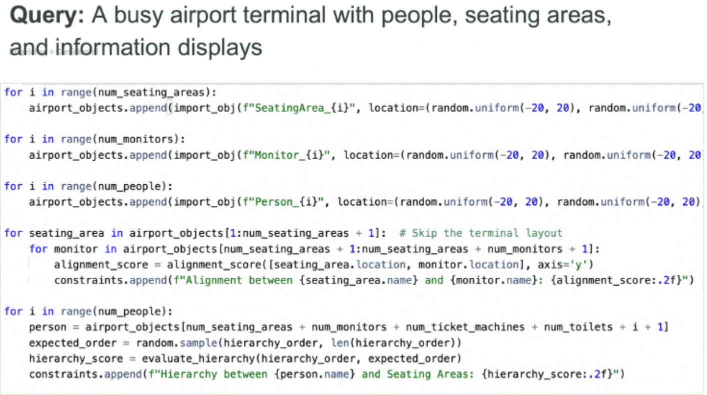




Query: Pedestrian crossroad intersection at right angle with people and cars

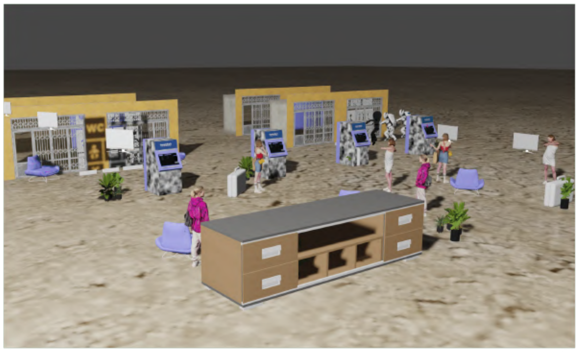
```
objects["GoldenGates"] = import_obj("GoldenGates", location=(0, 0, 0), orientation=(0, 0, 0))
objects["Skyway"] = import_obj("Skyway", location=(0, 0, 10), orientation=(0, 0, 0))
objects["Person1"] = import_obj("Person1", location=(-2, -2, 0), orientation=(0, 0, 0))
objects["Person2"] = import_obj("Person2", location=(2, 2, 0), orientation=(0, 0, 0))
objects["Car1"] = import_obj("Car1", location=(-3, 0, 0), orientation=(0, 0, 0))
objects["Car2"] = import_obj("Car2", location=(3, 0, 0), orientation=(0, 0, 0))
objects["GrandHotel"] = import_obj("GrandHotel", location=(0, 0, 0), orientation=(0, 0, 0))

# Apply constraints and store the results
alignment_gates_skyway = align_gates_skyway(objects)
proximity_person1_car1, proximity_person2_car2 = ensure_proximity_people_cars(objects)
alignment_stalls_x, alignment_stalls_y = align_stalls(objects)
perpendicular_skyway_main_road = ensure_perpendicular_skyway_main_road(objects)
walk_up_down_skyway_person1, walk_up_down_skyway_person2 = make_people_walk_up_down_skyway(objects)
cars_parallel_main_road_car1, cars_parallel_main_road_car2 = ensure_cars_parallel_main_road(objects)

def align_gates_skyway(objects):
    ......
```

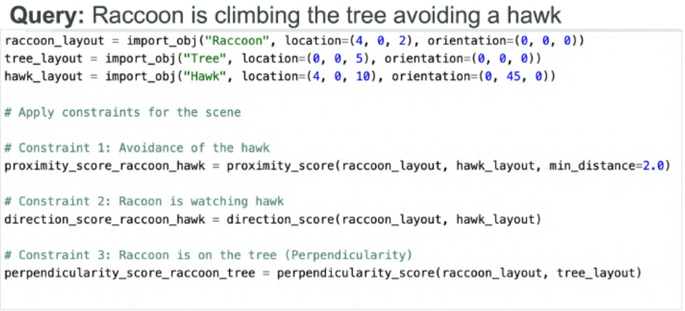




Query: A busy airport terminal with people, seating areas, and information displays

```
for i in range(num_seating_areas):
    airport_objects.append(import_obj(f"SeatingArea_{i}", location=(random.uniform(-20, 20), random.uniform(-20

for i in range(num_monitors):
    airport_objects.append(import_obj(f"Monitor_{i}", location=(random.uniform(-20, 20), random.uniform(-20, 20

for i in range(num_people):
    airport_objects.append(import_obj(f"Person_{i}", location=(random.uniform(-20, 20), random.uniform(-20, 20)

for seating_area in airport_objects[1:num_seating_areas + 1]:  # Skip the terminal layout
    for monitor in airport_objects[num_seating_areas + 1:num_seating_areas + num_monitors + 1]:
        alignment_score = alignment_score([seating_area.location, monitor.location], axis='y')
        constraints.append(f"Alignment between {seating_area.name} and {monitor.name}: {alignment_score:.2f}")

for i in range(num_people):
    person = airport_objects[num_seating_areas + num_monitors + num_ticket_machines + num_toilets + i + 1]
    expected_order = random.sample(hierarchy_order, len(hierarchy_order))
    hierarchy_score = evaluate_hierarchy(hierarchy_order, expected_order)
    constraints.append(f"Hierarchy between {person.name} and Seating Areas: {hierarchy_score:.2f}")
```





Query: Raccoon is climbing the tree avoiding a hawk

```
raccoon_layout = import_obj("Raccoon", location=(4, 0, 2), orientation=(0, 0, 0))
tree_layout = import_obj("Tree", location=(0, 0, 5), orientation=(0, 0, 0))
hawk_layout = import_obj("Hawk", location=(4, 0, 10), orientation=(0, 45, 0))

# Apply constraints for the scene

# Constraint 1: Avoidance of the hawk
proximity_score_raccoon_hawk = proximity_score(raccoon_layout, hawk_layout, min_distance=2.0)

# Constraint 2: Racoon is watching hawk
direction_score_raccoon_hawk = direction_score(raccoon_layout, hawk_layout)

# Constraint 3: Raccoon is on the tree (Perpendicularity)
perpendicularity_score_raccoon_tree = perpendicularity_score(raccoon_layout, tree_layout)
```

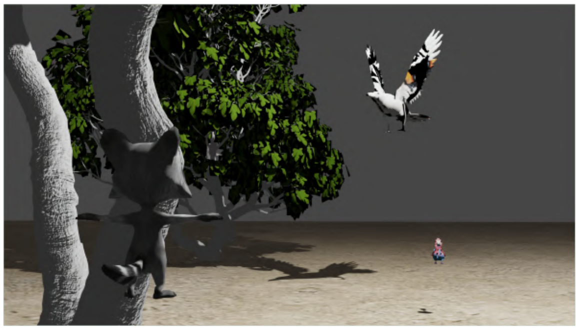


Figure 7. Examples of generated code and scenes

human intervention:

```
@dataclass
class Layout:
    location: Tuple[float, float, float]
    min: Tuple[float, float, float]
    max: Tuple[float, float, float]
    orientation: Tuple[float, float, float] # Euler angles (pitch, yaw, roll)

def scale_group(objects: List[bpy.types.Object], scale_factor: float) -> None:
    """
    Scale a group of objects by a given factor.

    Args:
        objects (List[bpy.types.Object]): List of Blender objects to scale.
        scale_factor (float): The scale factor to apply.

    Example:
        scale_group([object1, object2], 1.5)
    """
    for obj in objects:
        obj.scale = (obj.scale.x * scale_factor,
                     obj.scale.y * scale_factor,
                     obj.scale.z * scale_factor)
        obj.matrix_world = obj.matrix_world * scale_factor

def find_highest_vertex_point(objs: List[bpy.types.Object]) -> Dict[str, float]:
    """
    Find the highest vertex point among a list of objects.

    Args:
        objs (List[bpy.types.Object]): List of Blender objects to evaluate.

    Returns:
        Dict[str, float]: The lowest x, y, and z coordinates.

    Example:
        lowest_point = find_lowest_vertex_point([object1, object2])
    """
    bpy.context.view_layer.update()
    highest_points = {’x’: -float(’inf’), ’y’: -float(’inf’), ’z’: -float(’inf’)}

    for obj in objs:
        # Apply the object’s current transformation to its vertices
        obj_matrix_world = obj.matrix_world

        if obj.type == ’MESH’:
            # Update mesh to the latest data
            obj.data.update()
            for vertex in obj.data.vertices:
                world_vertex = obj_matrix_world @ vertex.co
                highest_points[’x’] = max(highest_points[’x’], world_vertex.x)
                highest_points[’y’] = max(highest_points[’y’], world_vertex.y)
                highest_points[’z’] = max(highest_points[’z’], world_vertex.z)

    return highest_points
```

E. Implementation Details

Canonicalization and Camera ViewPoint In the asset pool we use, all assets are not canonicalized originally, which indicates that if we import a person and a building, they can be of the same height. To address this issue, we add a preprocessing step to set each asset’s default height to be 1 (canonicalized), and then when the LLM retrieves an asset, it will predict its height to make it realistic in the scene. We didn’t do further height adjustment in the refinement procedure, so there are certain cases where some objects’ heights are actually wrong (say for a very flat carpet, the LLM is not able to correctly predict its thickness (height) at beginning).

The viewpoint (camera starting location, direction and view-point) are also predicted by the LLM in the final stage. Please see the “add_camera” function the LLM learns. We’re also exploring how to prompt it to generate the full camera trajectory to render a short video, which is an interesting future research direction of SceneCraft. Currently the lighting and ground texture are fixed. As seen in our visual examples, all grounds are set with standard sand texture, and we put a sun-like light at top. Enabling LLMs to adjust the rendering settings is definitely an interesting future research, especially when generating

```
def find_lowest_vertex_point(objs: List[bpy.types.Object]) -> Dict[str, float]:
    """
    Find the lowest vertex point among a list of objects.

    Args:
        objs (List[bpy.types.Object]): List of Blender objects to evaluate.

    Returns:
        Dict[str, float]: The lowest x, y, and z coordinates.

    Example:
        lowest_point = find_lowest_vertex_point([object1, object2])
    """
    bpy.context.view_layer.update()
    lowest_points = {’x’: float(’inf’), ’y’: float(’inf’), ’z’: float(’inf’)}

    for obj in objs:
        # Apply the object’s current transformation to its vertices
        obj_matrix_world = obj.matrix_world

        if obj.type == ’MESH’:
            # Update mesh to the latest data
            obj.data.update()
            for vertex in obj.data.vertices:
                world_vertex = obj_matrix_world @ vertex.co
                lowest_points[’x’] = min(lowest_points[’x’], world_vertex.x)
                lowest_points[’y’] = min(lowest_points[’y’], world_vertex.y)
                lowest_points[’z’] = min(lowest_points[’z’], world_vertex.z)

    return lowest_points

def calculate_shortest_distance(vertices1: Set[Tuple[float, float, float]],
                                vertices2: Set[Tuple[float, float, float]])
                                -> float:
    """
    Calculate the shortest distance between two sets of vertices.

    Args:
        vertices1 (Set[Tuple[float, float, float]]): First set of vertices.
        vertices2 (Set[Tuple[float, float, float]]): Second set of vertices.

    Returns:
        float: Shortest distance over the Z-axis.
    """
    min_distance = float(’inf’)
    for v1_tuple in vertices1:
        v1 = Vector(v1_tuple)
        for v2_tuple in vertices2:
            v2 = Vector(v2_tuple)
            distance = (v1 - v2).length
            min_distance = min(min_distance, distance)
    return min_distance
```

high-quality videos that require lighting changes.

Refinement Steps In our refinement algorithm, the average number of iterations is hard-coded as 4 steps without early stopping. We have tried different values and find having more steps can continue improving the per-scene performance. In our experiment, at the beginning of generation when there’s no good library, it require a lot of steps (¿10) to write proper codes as it needs to do everything from scratch; after we have a good learned library, the inner-loop only need to refine the constraint set and do some fine-tune, so normally 4 step can lead to a reasonable sub-scene planning. So we set 4 to strike a balance between result and efficiency, as 4-step normally is fine to get a reasonable sub-scene planning (given a good learned library).

Computational Cost Addressing more complex queries with a scalable solution is one of the design challenges of SceneCraft. In our design, we adopt a scene decomposition at the beginning, which separates the final task of a hundred assets into smaller-scale and simpler planning problems, each mostly only requiring 10-20 assets. For each sub-problem, we typically call the LLM 5 times, first to get a scene graph, and then 4 steps to iteratively update the scene. On average, these steps add up to consuming around 2k+ tokens (both input and output) The maximum number of subproblems of the current system is 7, so the maximum number of tokens for each scene design is around 15k, and the average is 6k. (translating to GPT pricing, using GPT4-turbo, it costs around 0.12$ for each query, and using GPT3.5, it costs around 0.006$ per query).

```
def rotate_objects_z_axis(objects: List[bpy.types.Object], angle_degrees: float) -> None:
    """
    Rotate a group of objects around the Z-axis by a given angle.

    Args:
        objects (List[bpy.types.Object]): List of objects to rotate.
        angle_degrees (float): The angle in degrees to rotate.

    Example:
        rotate_objects_z_axis([object1, object2], 45)
    """
    bpy.context.view_layer.update()
    angle_radians = math.radians(angle_degrees)  # Convert angle to radians
    rotation_matrix = mathutils.Matrix.Rotation(angle_radians, 4, ’Y’)
    lowest_point = find_lowest_vertex_point(objects)
    highest_points = find_highest_vertex_point(objects)
    center_point = {’x’: (lowest_point[’x’] + highest_points[’x’]) / 2,
                    ’y’: (lowest_point[’y’] + highest_points[’y’]) / 2,
                    ’z’: 0}
    for obj in objects:
        if obj.type == ’MESH’:
            obj.data.update()
            obj.matrix_world = obj.matrix_world @ rotation_matrix

    lowest_point = find_lowest_vertex_point(objects)
    highest_points = find_highest_vertex_point(objects)
    center_point[’x’] -= (lowest_point[’x’] + highest_points[’x’]) / 2
    center_point[’y’] -= (lowest_point[’y’] + highest_points[’y’]) / 2
    shift(objects, center_point)

def shift(objects: List[bpy.types.Object], shift_loc: Dict[str, float]) -> None:
    """
    Shift a group of objects with shift_loc.

    Args:
        objects (List[bpy.types.Object]): List of objects to rotate.
        shift_loc (float): The shift vector.

    Example:
        rotate_objects_z_axis([object1, object2], (5,3,1))
    """
    for obj in objects:
        # Shift object so the lowest point is at (0,0,0)
        obj.location.x += shift_loc[’x’]
        obj.location.y += shift_loc[’y’]
        obj.location.z += shift_loc[’z’]
    bpy.context.view_layer.update()
```

Asset retrieval is fast and computationally efficient thanks to efficient approximate-kNN methods. In our implementation now we use txtai (txt) to support embedding search.

Regarding scaling to complex design problems, we think in our pipeline, the cost and scale of each sub-problem should be similar (we observe that when the LLM tries to handle more than 20 assets during self-refinement, it does not give optimal results). Thus the key question is how to conduct better decomposition and grouping. For example, we can first generate nice housing, with different furniture and decorations, and use this as a grouped asset, to form a street, with slight changes for each house. There’s lots of exciting work of decomposing hard reasoning task into LLM-affordable sub-task including Parsel (Zelikman et al., 2023), Least-to-Most prompting (Zhou et al., 2023a), etc. With a proper decomposition (either in the current list implementation or in a tree-hierarchy), the computational cost will only scale linearly instead of exponentially w.r.t. number of assets in the scene. Thus we believe our system has the potential to handle much more complex questions with affordable cost.

F. Examples of annotated queries

Examples of the annotated queries as well as the per-scene scoring functions are shown in Figure 8 and Figure 10.

G. Prompt Used at each stage

The prompt used in SceneCraft is shown in Figure 10

```
def rotate_objects_z_axis(objects: List[bpy.types.Object], angle_degrees: float) -> None:
    """
    Rotate a group of objects around the Z-axis by a given angle.

    Args:
        objects (List[bpy.types.Object]): List of objects to rotate.
        angle_degrees (float): The angle in degrees to rotate.

    Example:
        rotate_objects_z_axis([object1, object2], 45)
    """
    bpy.context.view_layer.update()
    angle_radians = math.radians(angle_degrees)  # Convert angle to radians
    rotation_matrix = mathutils.Matrix.Rotation(angle_radians, 4, ’Y’)
    lowest_point = find_lowest_vertex_point(objects)
    highest_points = find_highest_vertex_point(objects)
    center_point = {’x’: (lowest_point[’x’] + highest_points[’x’]) / 2,
                    ’y’: (lowest_point[’y’] + highest_points[’y’]) / 2,
                    ’z’: 0}
    for obj in objects:
        if obj.type == ’MESH’:
            obj.data.update()
            obj.matrix_world = obj.matrix_world @ rotation_matrix

    lowest_point = find_lowest_vertex_point(objects)
    highest_points = find_highest_vertex_point(objects)
    center_point[’x’] -= (lowest_point[’x’] + highest_points[’x’]) / 2
    center_point[’y’] -= (lowest_point[’y’] + highest_points[’y’]) / 2
    shift(objects, center_point)

def shift(objects: List[bpy.types.Object], shift_loc: Dict[str, float]) -> None:
    """
    Shift a group of objects with shift_loc.

    Args:
        objects (List[bpy.types.Object]): List of objects to rotate.
        shift_loc (float): The shift vector.

    Example:
        rotate_objects_z_axis([object1, object2], (5,3,1))
    """
    for obj in objects:
        # Shift object so the lowest point is at (0,0,0)
        obj.location.x += shift_loc[’x’]
        obj.location.y += shift_loc[’y’]
        obj.location.z += shift_loc[’z’]
    bpy.context.view_layer.update()
```

```python
def check_vertex_overlap(vertices1: Set[Vector], vertices2: Set[Vector], threshold: float = 0.01) -> float:
    """
    Check if there is any overlap between two sets of vertices within a threshold.

    Args:
        vertices1 (Set[Vector]): First set of vertices.
        vertices2 (Set[Vector]): Second set of vertices.
        threshold (float): Distance threshold to consider as an overlap.

    Returns:
        bool: True if there is an overlap, False otherwise.
    """
    for v1_tuple in vertices1:
        v1 = Vector(v1_tuple)
        for v2_tuple in vertices2:
            v2 = Vector(v2_tuple)
            if (v1 - v2).length <= threshold:
                return 1.0
    return 0.0

def evaluate_constraints(assets, constraints):
    """Evaluate all constraints and return the overall score."""
    total_score = 0
    for constraint_func, involved_assets in constraints:
        # Assuming each constraint function takes involved assets and returns a score
        scores = constraint_func([assets[name] for name in involved_assets])
        total_score += sum(scores)  # Summing scores assuming each constraint can contribute multiple scores
    return total_score

def adjust_positions(assets, adjustment_step=0.1):
    """Randomly adjust the positions of assets."""
    for asset in assets.values():
        # Randomly adjust position within a small range to explore the space
        asset.location = (
            asset.location[0] + random.uniform(-adjustment_step, adjustment_step),
            asset.location[1] + random.uniform(-adjustment_step, adjustment_step),
            asset.location[2]  # Z position kept constant for simplicity
        )

def constraint_solving(assets, constraints, max_iterations=100):
    """Find an optimal layout of assets to maximize the score defined by constraints."""
    best_score = evaluate_constraints(assets, constraints)
    best_layout = {name: asset.copy() for name, asset in assets.items()}  # Assuming a copy method exists

    for _ in range(max_iterations):
        adjust_positions(assets)
        current_score = evaluate_constraints(assets, constraints)

        if current_score > best_score:
            best_score = current_score
            best_layout = {name: asset.copy() for name, asset in assets.items()}
        else:
            # Revert to best layout if no improvement
            assets = {name: layout.copy() for name, layout in best_layout.items()}

    return best_layout, best_score
```

```python
def normalize_vector(v: np.ndarray) -> np.ndarray:
    """Normalize a vector."""
    norm = np.linalg.norm(v)
    return v / norm if norm > 0 else np.zeros_like(v)

def orientation_similarity(orientation1: Tuple[float, float, float], orientation2: Tuple[float, float, float])
                           -> float:
    """Calculate the similarity between two orientations, represented as Euler angles."""
    # Convert Euler angles to vectors for simplicity in comparison
    vector1 = np.array(orientation1)
    vector2 = np.array(orientation2)
    # Calculate the cosine similarity between the two orientation vectors
    cos_similarity = np.dot(vector1, vector2) / (np.linalg.norm(vector1) * np.linalg.norm(vector2))
    return cos_similarity

def parallelism_score(assets: List[Layout]) -> float:
    """
    Evaluates and returns a score indicating the degree of parallelism in a list of assets’ layouts,
    considering both position and orientation.

    Args:
    assets (List[Layout]): A list of asset layouts.

    Returns:
    float: A score between 0 and 1 indicating the parallelism of the assets.
    """
    if len(assets) < 2:
        return 1.0  # Single asset or no asset is arbitrarily considered perfectly parallel

    # Positional parallelism
    vectors = [calculate_vector(assets[i].location, assets[i+1].location)
              for i in range(len(assets)-1)]
    normalized_vectors = [normalize_vector(v) for v in vectors]
    dot_products_position = [np.dot(normalized_vectors[i], normalized_vectors[i+1])
                       for i in range(len(normalized_vectors)-1)]

    # Rotational similarity
    orientation_similarities = [orientation_similarity(assets[i].orientation, assets[i+1].orientation)
             for i in range(len(assets)-1)]

    # Combine scores
    position_score = np.mean([0.5 * (dot + 1) for dot in dot_products_position])
    orientation_score = np.mean([(similarity + 1) / 2 for similarity in orientation_similarities])

    # Average the position and orientation scores for the final score
    final_score = (position_score + orientation_score) / 2

    return final_score


def calculate_distance(location1: Tuple[float, float, float], location2: Tuple[float, float, float]) -> float:
    """Calculate the Euclidean distance between two points."""
    return np.linalg.norm(np.array(location1) - np.array(location2))

def proximity_score(object1: Layout, object2: Layout, min_distance: float = 1.0, max_distance: float = 5.0) -> float:
    """
    Calculates a proximity score indicating how close two objects are, with 1 being very close and 0 being far apart.

    Args:
    object1 (Layout): The first object’s layout.
    object2 (Layout): The second object’s layout.
    min_distance (float): The distance below which objects are considered to be at optimal closeness. Scores 1.
    max_distance (float): The distance beyond which objects are considered too far apart. Scores 0.

    Returns:
    float: A score between 0 and 1 indicating the proximity of the two objects.
    """
    distance = calculate_distance(object1.location, object2.location)

    if distance <= min_distance:
        return 1.0
    elif distance >= max_distance:
        return 0.0
    else:
        # Linearly interpolate the score based on the distance
        return 1 - (distance - min_distance) / (max_distance - min_distance)
```

```python
def euler_to_forward_vector(orientation: Tuple[float, float, float]) -> np.ndarray:
    """Convert Euler angles to a forward direction vector."""
    pitch, yaw, _ = orientation
    # Assuming the angles are in radians
    x = np.cos(yaw) * np.cos(pitch)
    y = np.sin(yaw) * np.cos(pitch)
    z = np.sin(pitch)
    return np.array([x, y, z])

def calculate_vector(a: Tuple[float, float, float], b: Tuple[float, float, float]) -> np.ndarray:
    """Calculate the directional vector from point a to b."""
    return np.array(b) - np.array(a)

def direction_score(object1: Layout, object2: Layout) -> float:
    """
    Calculates a score indicating how directly object1 is targeting object2.

    Args:
    object1 (Layout): The first object’s layout, assumed to be the one doing the targeting.
    object2 (Layout): The second object’s layout, assumed to be the target.

    Returns:
    float: A score between 0 and 1 indicating the directionality of object1 towards object2.
    """
    forward_vector = euler_to_forward_vector(object1.orientation)
    target_vector = calculate_vector(object1.location, object2.location)
    # Normalize vectors to ensure the dot product calculation is based only on direction
    forward_vector_normalized = normalize_vector(forward_vector)
    target_vector_normalized = normalize_vector(target_vector)
    # Calculate the cosine of the angle between the two vectors
    cos_angle = np.dot(forward_vector_normalized, target_vector_normalized)
    # Map the cosine range [-1, 1] to a score range [0, 1]
    score = (cos_angle + 1) / 2
    return score

def alignment_score(assets: List[Layout], axis: str) -> float:
    """
    Calculates an alignment score for a list of assets along a specified axis.

    Args:
    assets (List[Layout]): A list of asset layouts to be evaluated for alignment.
    axis (str): The axis along which to evaluate alignment (’x’, ’y’, or ’z’).

    Returns:
    float: A score between 0 and 1 indicating the degree of alignment along the specified axis.
    """
    if not assets or axis not in [’x’, ’y’, ’z’]:
        return 0.0  # Return a score of 0 for invalid input

    # Axis index mapping to the location tuple
    axis_index = {’x’: 0, ’y’: 1, ’z’: 2}[axis]

    # Extract the relevant coordinate for each asset based on the chosen axis
    coordinates = [asset.location[axis_index] for asset in assets]
    # Calculate the variance of these coordinates
    variance = np.var(coordinates)
    # Inverse the variance to calculate the score, assuming a lower variance indicates better alignment
    # Normalize the score to be between 0 and 1, considering a reasonable threshold for "perfect" alignment
    threshold_variance = 1.0  # Define a threshold variance for "perfect" alignment
    score = 1 / (1 + variance / threshold_variance)
    # Clamp the score between 0 and 1
    score = max(0, min(score, 1))
    return score

def check_vertex_overlap(vertices1: Set[Vector], vertices2: Set[Vector], threshold: float = 0.01) -> float:
    """
    Check if there is any overlap between two sets of vertices within a threshold.

    Args:
        vertices1 (Set[Vector]): First set of vertices.
        vertices2 (Set[Vector]): Second set of vertices.
        threshold (float): Distance threshold to consider as an overlap.

    Returns:
        bool: True if there is an overlap, False otherwise.
    """
    for v1_tuple in vertices1:
        v1 = Vector(v1_tuple)
        for v2_tuple in vertices2:
            v2 = Vector(v2_tuple)
            if (v1 - v2).length <= threshold:
                return 0.0
    return 1.0
```

```
def symmetry_score(assets: List[Layout], axis: str) -> float:
    """
    Calculates a symmetry score for a list of assets along a specified axis.

    Args:
    assets (List[Layout]): A list of asset layouts to be evaluated for symmetry.
    axis (str): The axis along which to evaluate symmetry (’x’, ’y’, or ’z’).

    Returns:
    float: A score between 0 and 1 indicating the degree of symmetry along the specified axis.
    """
    if not assets or axis not in [’x’, ’y’, ’z’]:
        return 0.0  # Return a score of 0 for invalid input

    # Axis index mapping to the location tuple
    axis_index = {’x’: 0, ’y’: 1, ’z’: 2}[axis]

    # Find the median coordinate along the specified axis to define the symmetry axis
    coordinates = [asset.location[axis_index] for asset in assets]
    symmetry_axis = np.median(coordinates)

    # Calculate the deviation from symmetry for each asset
    deviations = []
    for asset in assets:
        # Find the mirrored coordinate across the symmetry axis
        mirrored_coordinate = 2 * symmetry_axis - asset.location[axis_index]
        # Find the closest asset to this mirrored coordinate
        closest_distance = min(abs(mirrored_coordinate - other.location[axis_index]) for other in assets)
        deviations.append(closest_distance)

    # Calculate the average deviation from perfect symmetry
    avg_deviation = np.mean(deviations)

    # Convert the average deviation to a score, assuming smaller deviations indicate better symmetry
    # The scoring formula can be adjusted based on the specific requirements for symmetry in the application
    max_deviation = 10.0  # Define a maximum deviation for which the score would be 0
    score = max(0, 1 - avg_deviation / max_deviation)

    return score

def perpendicularity_score(object1: Layout, object2: Layout) -> float:
    """
    Calculates a score indicating how perpendicular two objects are, based on their forward direction vectors.

    Args:
    object1 (Layout): The first object’s layout, including its orientation as Euler angles.
    object2 (Layout): The second object’s layout, including its orientation as Euler angles.

    Returns:
    float: A score between 0 and 1 indicating the degree of perpendicularity.
    """
    vector1 = euler_to_forward_vector(object1.orientation)
    vector2 = euler_to_forward_vector(object2.orientation)
    cos_angle = np.dot(vector1, vector2) / (np.linalg.norm(vector1) * np.linalg.norm(vector2))
    score = 1 - np.abs(cos_angle)
    return score

def calculate_volume(layout: Layout) -> float:
    """Calculate the volume of an object based on its layout dimensions."""
    length = abs(layout.max[0] - layout.min[0])
    width = abs(layout.max[1] - layout.min[1])
    height = abs(layout.max[2] - layout.min[2])
    return length * width * height
```

```python
def evaluate_hierarchy(assets: List[Layout], expected_order: List[str]) -> float:
    """
    Evaluates how well a list of objects follows a specified hierarchical order based on size.

    Args:
    assets (List[Layout]): A list of asset layouts to be evaluated.
    expected_order (List[str]): A list of identifiers (names) for the assets, specifying the expected order of sizes.

    Returns:
    float: A metric indicating how well the actual sizes of the objects match the expected hierarchical order.
    """
    # Map identifiers to volumes
    id_to_volume = {asset_id: calculate_volume(asset) for asset_id, asset in zip(expected_order, assets)}

    # Calculate the actual order based on sizes
    actual_order = sorted(id_to_volume.keys(), key=lambda x: id_to_volume[x], reverse=True)

    # Evaluate the match between the expected and actual orders
    correct_positions = sum(1 for actual, expected in zip(actual_order, expected_order) if actual == expected)
    total_positions = len(expected_order)

    # Calculate the match percentage as a measure of hierarchy adherence
    match_percentage = correct_positions / total_positions

    return match_percentage


def calculate_angle_from_center(center: Tuple[float, float, float], object_location: Tuple[float, float, float])
                               -> float:
    """Calculate the angle of an object relative to a central point."""
    vector = np.array(object_location) - np.array(center)
    angle = np.arctan2(vector[1], vector[0])
    return angle

def rotation_uniformity_score(objects: List[Layout], center: Tuple[float, float, float]) -> float:
    """
    Calculates how uniformly objects are distributed around a central point in terms of rotation.

    Args:
    objects (List[Layout]): A list of object layouts to be evaluated.
    center (Tuple[float, float, float]): The central point around which objects are rotating.

    Returns:
    float: A score between 0 and 1 indicating the uniformity of object distribution around the center.
    """
    angles = [calculate_angle_from_center(center, obj.location) for obj in objects]
    angles = np.sort(np.mod(angles, 2*np.pi))  # Normalize angles to [0, 2\pi] and sort

    # Calculate differences between consecutive angles, including wrap-around difference
    angle_diffs = np.diff(np.append(angles, angles[0] + 2*np.pi))

    # Evaluate uniformity as the variance of these differences
    variance = np.var(angle_diffs)
    uniformity_score = 1 / (1 + variance)  # Inverse variance, higher score for lower variance

    return uniformity_score

def put_ontop(obj_dict, moving_set_name, target_set_name, threshold, step):
    """
    Adjust objects in moving_set_name until the shortest distance to target_set_name is below the threshold.

    Args:
        obj_dict (dict): Dictionary of object sets.
        moving_set_name (str): The key for the set of objects to move.
        target_set_name (str): The key for the set of objects to calculate distance to.
        threshold (float): The distance threshold.
        step (float): The step by which to move objects in the Z direction.
    """
    while True:
        vertices_set1 = get_all_vertices(obj_dict[moving_set_name])
        vertices_set2 = get_all_vertices(obj_dict[target_set_name])
        shortest_distance = calculate_shortest_distance(vertices_set1, vertices_set2)
        print(shortest_distance)

        if shortest_distance < threshold:
            break

        for obj in obj_dict[moving_set_name]:
            obj.location.z -= max(step, shortest_distance)

        bpy.context.view_layer.update()
```

```python
def repeat_object(original: Layout, direction: Tuple[float, float, float], repetitions: int, distance: float)
                 -> List[Layout]:
    """
    Creates a series of duplicated objects, repeating them in a specified direction at a set distance.

    Args:
    original (Layout): The original object to be repeated.
    direction (Tuple[float, float, float]): The direction vector along which to repeat the object.
    repetitions (int): The number of times the object should be repeated.
    distance (float): The distance between each object.

    Returns:
    List[Layout]: A list of Layout objects representing the original and its duplicates.
    """
    repeated_objects = [original]  # Include the original object in the output list

    for i in range(1, repetitions):
        # Calculate the new location for each repeated object
        new_location = (
            original.location[0] + direction[0] * distance * i,
            original.location[1] + direction[1] * distance * i,
            original.location[2] + direction[2] * distance * i
        )
        # Create a new Layout instance for each repetition
        new_object = Layout(
            location=new_location,
            min=original.min,
            max=original.max,
            orientation=original.orientation
        )
        repeated_objects.append(new_object)

    return repeated_objects

def add_camera(location: Tuple[float, float, float], target_point: Tuple[float, float, float], lens: float = 35)
          -> bpy.types.Object:
    """
    Add a camera to the Blender scene.

    Args:
        location (Vector): The location to place the camera.
        target_point (Vector): The point the camera should be aimed at.
        lens (float, optional): The lens size. Defaults to 35.

    Returns:
        bpy.types.Object: The created camera object.

    Example:
        camera = add_camera((10, 10, 10), (0, 0, 0))
    """
    # Create a new camera data object
    cam_data = bpy.data.cameras.new(name="Camera")
    cam_data.lens = lens  # Set the lens property

    # Create a new camera object and link it to the scene
    cam_object = bpy.data.objects.new('Camera', cam_data)
    bpy.context.collection.objects.link(cam_object)

    # Set the camera location
    cam_object.location = location

    # Calculate the direction vector from the camera to the target point
    direction = Vector(target_point) - Vector(location)
    # Orient the camera to look at the target point
    rot_quat = direction.to_track_quat('-Z', 'Y')
    cam_object.rotation_euler = rot_quat.to_euler()

    # Set the created camera as the active camera in the scene
    bpy.context.scene.camera = cam_object

    return cam_object
```

```python
scene_1 = {
    "description": "A book lying flat on a table, two chair on each side",
    "assets": ["book", "table"],
    "relationships": {
        "relativity": {
            "description": "The book should be on top of the table",
            "involved_objects": ["book", "table"]
        },
        "alignment": {
            "description": "The book should be aligned with the table in the x and y directions",
            "involved_objects": ["book", "table"]
        }
    }
}

def score_1(locs):
    # Extracting locations
    x_book, y_book, z_book = locs[’book’][’x’], locs[’book’][’y’], locs[’book’][’z’]
    x_table, y_table, z_table = locs[’table’][’x’], locs[’table’][’y’], locs[’table’][’z’]
    # Relativity score (penalizing if book is below table surface)
    relativity_score = max(0, z_table - z_book)  # positive if book is below table
    # Alignment score (difference in x and y positions, zero if perfectly aligned)
    alignment_score_x = abs(x_book - x_table)
    alignment_score_y = abs(y_book - y_table)
    # Total score (sum of individual scores)

    total_score = relativity_score + alignment_score_x + alignment_score_y
    return 1 -  total_score / 100.

scene_2 = {
    "description": "A busy airport terminal with people, seating areas, and information displays",
    "assets": ["person1", "person2", "seating_area", "information_display"],
    "relationships": {
        "grouping": {
            "description": "People should be grouped near the seating areas",
            "involved_objects": ["person1", "person2", "seating_area"]
        },
        "alignment": {
            "description": "Information displays should be aligned above the seating areas",
            "involved_objects": ["seating_area", "information_display"]
        },
        "proximity": {
            "description": "People should be close to information displays for visibility",
            "involved_objects": ["person1", "person2", "information_display"]
        }
    }
}
def score_2(locs):
    def distance(a, b):
        return math.sqrt((a[’x’] - b[’x’])**2 + (a[’y’] - b[’y’])**2 + (a[’z’] - b[’z’])**2)

    # Grouping score (distance of people from seating areas)
    grouping_score = sum(distance(locs[p], locs[’seating_area’]) for p in [’person1’, ’person2’])
    # Alignment score (information display above seating areas)
    alignment_score = abs(locs[’seating_area’][’y’] - locs[’information_display’][’y’])
    # Proximity score (people close to information displays)
    proximity_score = sum(distance(locs[p], locs[’information_display’]) for p in [’person1’, ’person2’])

    # Total score
    total_score = grouping_score + alignment_score + proximity_score
    return 1 - total_score / 100.
```

Figure 8. Example of annotated queries and scoring function

```
scene_3 = {
    "description": "Three boxes of different sizes, stacked on top of each other",
    "assets": ["box1", "box2", "box3"],
    "relationships": {
        "hierarchy": {
            "description": "The boxes should be in descending order of size from bottom to top",
            "involved_objects": ["box1", "box2", "box3"]
        },
        "layering": {
            "description": "The boxes should be placed one above the other",
            "involved_objects": ["box1", "box2", "box3"]
        }
    },
}

def score_3(locs):
    # Extracting locations
    z_box1, z_box2, z_box3 = locs[’box1’][’z’], locs[’box2’][’z’], locs[’box3’][’z’]
    w_box1, w_box2, w_box3 = locs[’box1’][’w’], locs[’box2’][’w’], locs[’box3’][’w’]
    # Hierarchy score (sizes)
    hierarchy_score = 0
    if not (w_box1 > w_box2 > w_box3):
        hierarchy_score = abs(w_box1 - w_box2) + abs(w_box2 - w_box3)
    # Layering score (z-axis positioning)
    layering_score = 0
    if not (z_box1 < z_box2 < z_box3):
        layering_score = abs(z_box1 - z_box2) + abs(z_box2 - z_box3)
    # Total score
    total_score = hierarchy_score + layering_score
    return 1 - total_score / 100

scene_4 = {
    "description": "A new solar system with planets orbiting around a small star",
    "assets": ["sun", "planet1", "planet2", "planet3"],
    "relationships": {
        "rotation": {
            "description": "Planets should orbit around the sun",
            "involved_objects": ["planet1", "planet2", "planet3", "sun"]
        },
        "scaling": {
            "description": "Planets should vary in size",
            "involved_objects": ["planet1", "planet2", "planet3"]
        }
    }
}

def score_4(locs):
    import math
    def distance(a, b):
        return math.sqrt((a[’x’] - b[’x’])**2 + (a[’y’] - b[’y’])**2 + (a[’z’] - b[’z’])**2)

    # Rotation score (distance from sun)
    rotation_score = sum(distance(locs[p], locs[’sun’]) for p in [’planet1’, ’planet2’, ’planet3’])

    # Scaling score (size of planets)
    scaling_score = abs(locs[’planet1’][’size’] - locs[’planet2’][’size’]) +
                    abs(locs[’planet2’][’size’] - locs[’planet3’][’size’])

    # Total score
    total_score = rotation_score + scaling_score
    return 1 - total_score / 100
```

Figure 9. Example of annotated queries and scoring function

```
query_find_assets = """I am writing several blender scripts to generate scenes for: %s.
Please think step by step and then give me the assets (each is a single unit, avoid a composite set that
contains multiple objects) that shall appear in these scenes.
After explanation, structured in: Output: 1) x1: y1; 2) x2: y2; 3) ...
Each with a general descriptive name (x) and a very detailed visual description (y)."""


query_height_assets = """I am writing several blender scripts to generate scenes for %s.
Below are the assets we’d like to use. Now we need to scale them to correct height, please generate a python
dictionary called height_dict, where key is each asset’s name, and value
is a number representing the height (measured in metre)

%s

Output the complete python dict via height_dict = {asset_name: height, ...},
also give detailed explanation as comment before the value in the dict.
"""


query_plan_assets = """I am writing several blender scripts to generate a scene for %s.

Below are the assets I’d like to use:
%s

Now I want a concrete plan to put them into the scene. Please think step by step, and give a multi-step plan to put
assets into the scene. For each step, structure your output as:
layout_plan_i = {
    "title": title_i,
    "asset_list" :
        [asset_name_1, asset_name_2],
    "description": desc_i
}
where title_i is the high-level name for this step, and desc is detailed visual text description of what it shall
look like after layout. asset_list is the non-empty list of assets to be added in this step.
Please think step by step, place assets from environmental ones to more details assets. Return me a list of
python dictonaries layout_plan_1, layout_plan_2, ...
"""

prompt_graph = """
You are tasked with constructing a relational bipartite graph for a 3D scene based on the provided description and
asset list. Your goal is to identify the spatial and contextual relationships
between assets and represent these relationships in a structured format. Follow these steps:

1. Review the scene description and the list of assets.
2. Determine the spatial and contextual relationships needed to accurately represent the scene’s layout. Consider
relationships like proximity, alignment, parallelism, etc.
3. Construct the relational bipartite graph ‘G(s) = (A, R, E)‘ where:
   - ‘A‘ represents the set of assets.
   - ‘R‘ represents the set of relations as nodes.
   - ‘E‘ represents the edges connecting a relation node to a subset of assets ‘E(r)‘ in the scene.
4. For each identified relationship, create a relation node and link it to the  assets through edges in the graph.

Output your findings in a structured format:
- List of relation nodes ‘R‘ with their types and descriptions.
- Edges ‘E‘ that link assets to their corresponding relation nodes.

This process will guide the arrangement of assets in the 3D scene, ensuring they are positioned, scaled, and oriented
correctly according to the scene’s requirements and the relationships between objects.
"""
```

Figure 10. Example of prompts being used

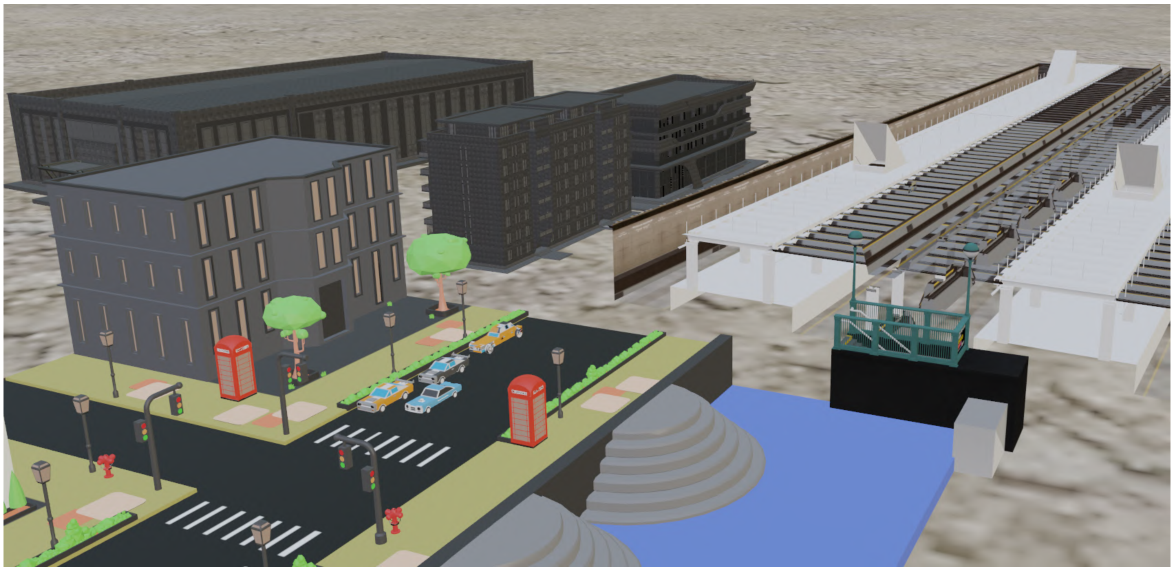


Figure 11. City scene synthesized by SceneCraft. In this design, there already exists an asset with a basic cross-road, and SceneCraft is able to put different cars in it, trees and lamps beside the road, and put the buildings on the other side; the overall structure looks good showing it understands the basic structure of the road.

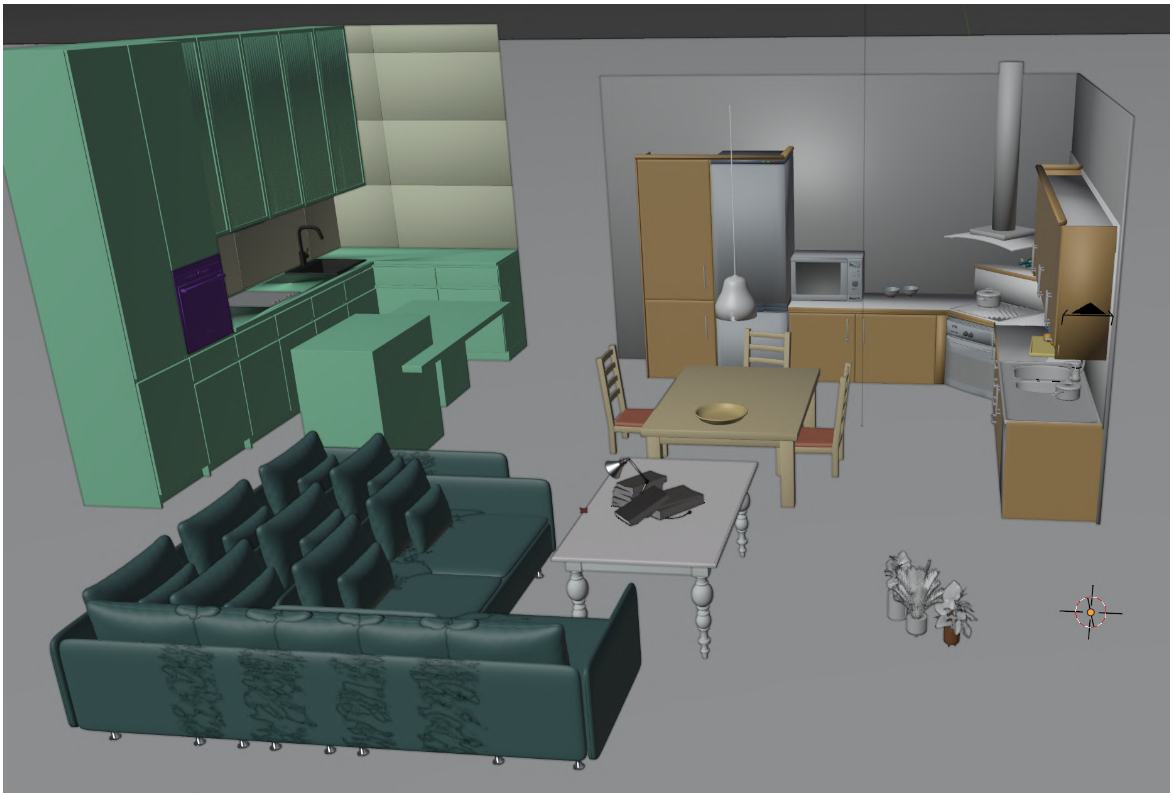


Figure 12. Indoor scene (Blender) synthesized by SceneCraft

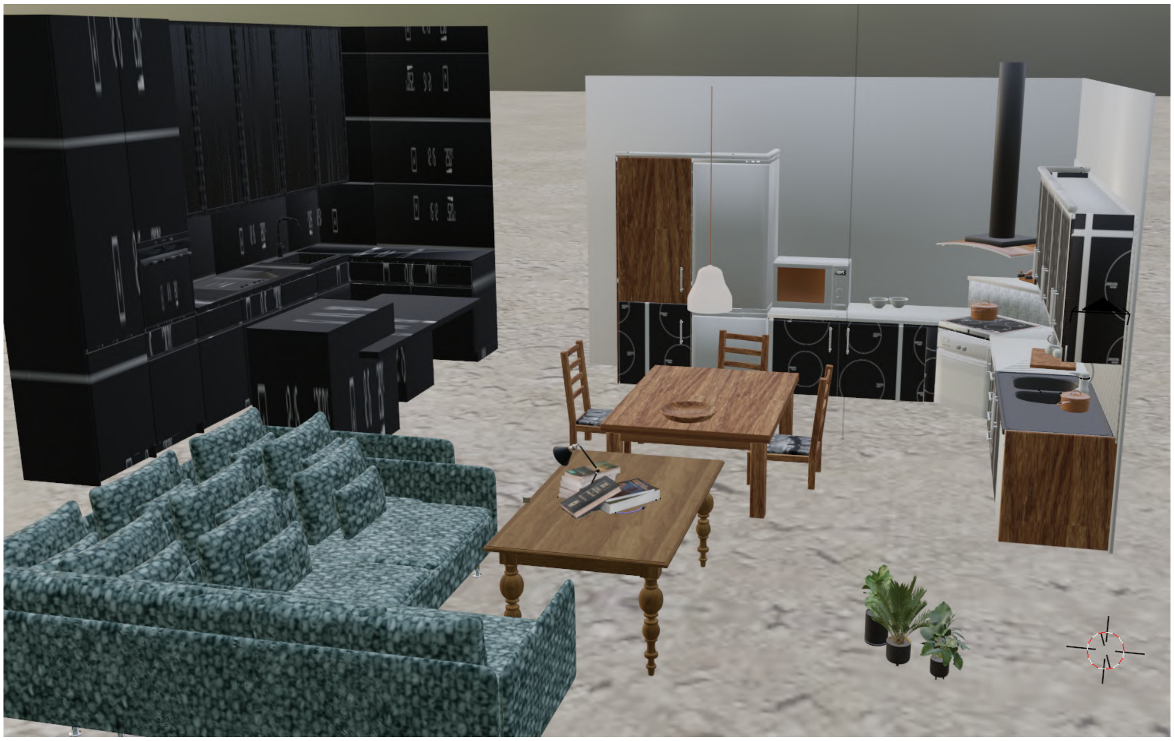
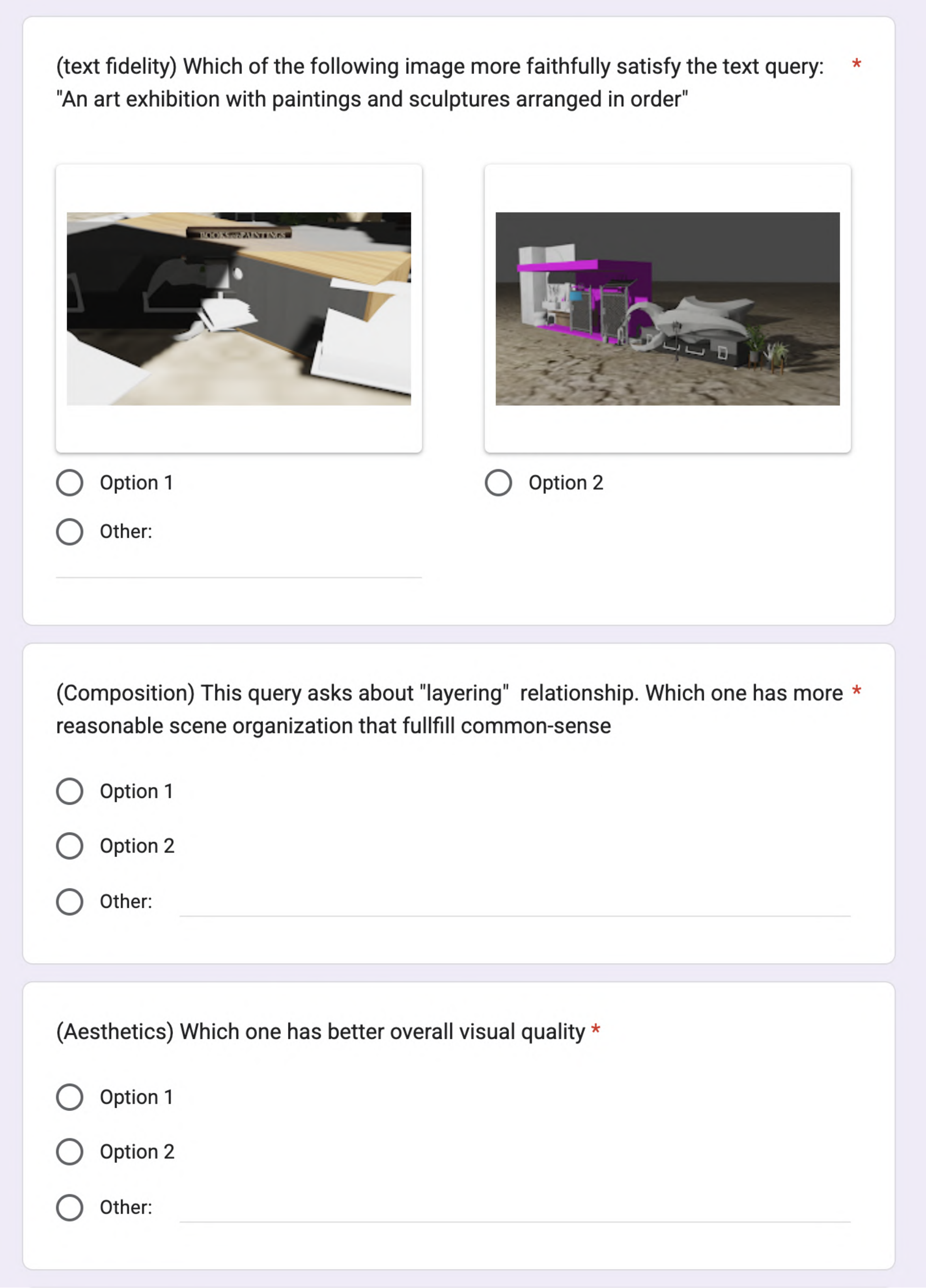


Figure 13. Indoor scene (Renderred) synthesized by SceneCraft. It learns to align to table, putting chandelier over one table, putting three chairs around the main table, book, lamp and pen on another table, make two table in an aligned array, and adjusting the two kitchen bundle to surround the table and main living space, which shows that SceneCraft can understand these spatial constraints.



(text fidelity) Which of the following image more faithfully satisfy the text query: "An art exhibition with paintings and sculptures arranged in order" *

○ Option 1

○ Option 2

○ Other:

(Composition) This query asks about "layering" relationship. Which one has more reasonable scene organization that fullfill common-sense *

○ Option 1

○ Option 2

○ Other:

(Aesthetics) Which one has better overall visual quality *

○ Option 1

○ Option 2

○ Other:

Figure 14. Questionnaire Interface, with three questions, about 1) Text Fidelity; 2) Composition; 3) Aesthetics